UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Jure Cetina

Analiza učinkovitosti mutacij mutacijskega operatorja AOR

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2014

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Jure Cetina

Analiza učinkovitosti mutacij mutacijskega operatorja AOR

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: viš. pred. dr. Igor Rožanc

Ljubljana, 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Naslov: Analiza učinkovitosti mutacij mutacijskega operatorja AOR

Tematika naloge:

Mutacijsko testiranje je uveljavljen testni pristop, s katerim lahko preverimo kakovost obstoječe množice testnih primerov. Njegova uporaba v praksi zahteva premišljeno izbiro mutacijskih operatorjev, saj sicer število mutantov preveč naraste in zahteva preveč časa za izvedbo.
V diplomski nalogi najprej celovito predstavite področje mutacijskega testiranja ter mutacijsko orodje MuJava. Z njim izvedite eksperiment za preverjanje učinkovitosti mutacij mutacijskega operatorja AOR, ki opredeljuje mutacijo kot zamenjavo enega aritmetičnega operatorja z drugim. Analizo opravite tako z vidika učinkovitosti posameznih aritmetičnih operatorjev kot z vidika večkratne uporabe tovrstnih mutacij.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani **Jure Cetina**, z vpisno številko **63110056**, sem avtor diplomskega dela z naslovom:

Analiza učinkovitosti mutacij mutacijskega operatorja AOR

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom **viš. pred. dr. Igorja Rožanca**,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne _____ Podpis avtorja: _____

Zahvale bi namenil svojemu mentorju viš. pred. dr. Igorju Rožancu za pomoč pri izvajanju eksperimenta in izdelavi diplomskega dela ter za prijetno novo-pridobljeno izkušnjo.

Prav tako se zahvaljujem vsem, ki so mi v času izdelave diplomskega dela stali ob strani in mi popestrili dneve.

Kazalo

Kazalo slik

Kazalo tabel

Slovar pojmov

Testiranje programske opreme (angl. software testing): je vrednotenje programske opreme na podlagi opazovanja izvajanja le-te. [31]

Testni primer (angl. test case): je sestavljen iz zbirke vhodnih podatkov, pričakovanih izhodnih rezultatov ter vrednosti predpon in končnic, ki so potrebne za izvedbo in oceno testirane programske opreme. [31]

Pokritje programske kode (angl. code coverage): je merilo, ki opisuje stopnjo do katere je izvorna koda programa testirana z določenim naborom testnih primerov.

Kriterij pokritja (angl. coverage criterion): je pravilo ali množica pravil, ki določa testne zahteve za testno množico. [31]

Napaka (angl. fault): je pomanjkljivost (pomota) v programu. Napake običajno vnesejo razvijalci z napačnim razumevanjem zahtev. [31]

Okvara (angl. error) je neveljavno stanje, ki je posledica napake. Do okvare pride, ko se izvede del programa, ki vsebuje napako. [31]

Odpoved (angl. failure) odstopanje od zahtevanega delovanja. Okvara preide v odpoved v primeru, ko ne dobimo pričakovanega rezultata. Vsaka okvara ne povzroči odpovedi. [31]

Sintaksa (angl. syntax): računalniškega jezika je nabor pravil, ki definirajo kombinacije simbolov. Posamezna kombinacija simbolov predstavlja pravilno strukturiran fragment znotraj določenega jezika.

Semantika (angl. semantics): računalniškega jezika definira pomen sintakse le-tega.

Operator: je konstrukt, ki se obnaša enako kot funkcija, vendar se sintaktično ali semantično razlikuje od le-teh.

Povzetek

Diplomska naloga zajema področje testiranja programske opreme z izvajanjem mutacijske analize. Mutacijska analiza oziroma testiranje se uporablja za ustvarjanje novih ali za ovrednotenje že obstoječih programskih testov. Kakovost testov se določi na način, da se v program vstavijo vnaprej definirane napake. V kolikor testi v programu odkrijejo vse napake, so učinkoviti, v nasprotnem primeru pa so testi neučinkoviti in jih je potrebno ustrezno dopolniti.

V diplomski nalogi je tako podrobneje predstavljen koncept mutacijskega testiranja, postopek generiranja in testiranja mutantov z uporabo sistema MuJava in eksperiment, ki se osredotoča na aritmetični mutacijski operator. Eksperiment naslavlja problem časovne zahtevnosti mutacijskega testiranja in s pomočjo analize učinkovitosti posameznih mutacij mutacijskega operatorja AOR predlaga možne rešitve na področju optimizacije postopka generiranja mutantov.

Ključne besede: mutacijsko testiranje, MuJava, mutanti, mutacije, mutacijski operator, AOR.

Abstract

The thesis covers the field of software testing with the use of mutation testing. Mutation testing or mutation analysis is used to create new software tests or to evaluate the efficiency of preexisting tests. The quality of a given test is determined by injecting predefined faults into a program's source code. If the test detects faults in the program it is considered to be adequate. In contrast, if the test does not detect any inappropriate behavior it is considered to be inadequate and must therefore be adjusted.

The thesis describes the concept of mutation testing in detail, demonstrates the process of generating and testing mutants with the use of the mutation testing system called MuJava and carries out an experiment that focuses on the arithmetic mutation operator. The experiment addresses the problem of runtime complexity of mutation testing and proposes possible optimization solutions in the process of generating mutants. The solutions are based on the analysis that determines the efficiency of individual mutations that are applied as part of the AOR mutation operator.

Keywords: mutation testing, MuJava, mutants, mutations, mutation operator, AOR.

Poglavje 1 Uvod

Testiranje programske opreme je pomemben, a finančno potraten del procesa razvoja programske opreme. Zato je še toliko pomembnejša izbira pravilnega pristopa k testiranju, ki zagotavlja večjo avtomatizacijo in bolj optimalno porabo sredstev.

Mutacijska analiza je testni kriterij, ki uporabnikom pomaga zasnovati zelo učinkovite testne primere, s katerimi lahko zagotovijo pravilno delovanje programske opreme. Mutacijski operatorji nad izvornim programom izvedejo sintaktične spremembe, ki vplivajo na obnašanje programa in na ta način generirajo mutante. Kakovost testov se določi na podlagi tega, koliko mutantov testi "ubijejo" - koliko sprememb oziroma mutantov testi prepoznajo in jih označijo kot nepravilno delovanje programa. Učinkovitost mutacije je neposredno odvisna od uporabljenih mutacijskih operatorjev in od njenih stroškov. Do potratnosti pride, ko mutacijski operatorji proizvedejo ogromno število mutantov, saj mora biti vsak mutant testiran, kar zahteva veliko računskega časa. Poleg tega morajo uporabniki analizirati preživete mutante in na podlagi le-teh nadgraditi oziroma dopisati testne primere, ki jih bodo ubili. Ravno zaradi perečega problema potratnosti je mutacijsko testiranje le v manjšem obsegu razširjeno izven raziskovalnega in akademskega okolja. Številni raziskovalci tako razvijajo razne nove prijeme na področju optimizacije, da bi mutacijsko analizo približali strokovnim uporabnikom in omogočili uporabo na realnih primerih.

V diplomski nalogi je v prvem delu podrobneje opisan koncept mutacijskega testiranja. Prvi del tako zajema razlage pojmov in postopkov, opisuje slabosti mutacijskega testiranja in možne rešitve aktualnih problemov, prav tako pa se poglobi v sestavo in funkcionalnosti mutacijskih operatorjev, ki so bili osrednji del naše raziskave. Drugi del je namenjen predstavitvi sistema za izvajanje mutacijskega testiranja MuJava, ki smo ga uporabili pri realizaciji našega eksperimenta. Po korakih je opisana implementacija sistema, opisane so funkcionalnosti posameznega dela sistema, poglavje pa se zaključi s primerjavo sistema MuJava z ostalimi sistemi, ki so bili prav tako možna orodja za izvajanje našega eksperimenta. Zadnji del opisuje potek naše raziskave in dobljene rezultate. Bralca seznani s tematiko eksperimenta, opiše testne komponente in okoliščine, predstavi in obrazloži rezultate, za zaključek pa povzame ugotovitve, ki so razvidne iz rezultatov.

Poglavje 2 Mutacijsko testiranje

2.1 Mutacijsko testiranje

Mutacijsko testiranje je ena izmed mnogih metodologij za testiranje programske opreme in je bila prvič predstavljena že v sedemdesetih letih prejšnjega stoletja. Mutiranje na splošno velja za najmočnejši testni kriterij v smislu odkrivanja velikega števila napak. Medtem ko so drugi prijemi za testiranje usmerjeni na pravilno delovanje programov, se mutacijsko testiranje osredotoča na testne primere, ki so uporabljeni za testiranje programov.

Glavna ideja je ustvarjanje dobrega nabora testnih primerov. Testiranje programov je tako le stranski učinek ustvarjanja kakovostnih testov. Dobri testi so testi, ki so zmožni odkriti tako lahko kot težko najdljive napake v programu. Za testiranje je zato potrebno v program namenoma vstaviti napake, ki jih programerji pogosto naredijo in s tem ugotoviti, ali jih testi odkrijejo ali ne. Tradicionalno mutacijsko testiranje se usmerja na napake, ki so najbližje pravilni različici programa oziroma program spremenijo na zelo majhen način. Te majhne napake domnevno simulirajo vse možne napake zaradi naslednjih dveh hipotez:

- Hipoteza "Sposoben programer" (angl. Competent Programer Hypothesis ali krajše CPH) navaja, da so programerji sposobni in tako razvijajo programe, ki so po vsebini in izvedbi zelo podobni pravilni obliki programa. Posledično je večina napak le manjše sintaktično odstopanje, ki vodi k nepravilnemu obnašanju programa. Hipotezo je leta 1978 predstavil Richard DeMillo in drugi. [1], [12]

- Druga hipoteza se imenuje "Učinek združevanja" (angl. Coupling Effect) in jo je prav tako zasnoval Richard DeMillo in drugi. Hipoteza navaja, da kompleksne napake nastanejo iz kombinacij preprostejših napak in da so testi, ki zaznajo preproste napake, dovolj občutljivi, da zaznajo tudi kompleksnejše napake. [12]

Mutacijska analiza je sestavljena iz dveh korakov. Prvi korak zajema ustvarjanje napačnih različic programa. Proces vstavljanja napak (angl. fault seeding) se imenuje *mutacija*, nastal produkt oziroma program s spremenjeno kodo pa *mutant*. Večina mutacijskih orodij ustvari mutante, ki vsebujejo le eno napako in se imenujejo *mutanti prvega reda* (angl. single-order mutants). Za bolj podrobno analizo pa je možno ustvariti tudi mutante, ki vsebujejo več sprememb oziroma napak. Tovrstni mutanti se imenujejo *mutanti višjega reda*

(angl. higher-order mutants). Pravilo o preoblikovanju, ki opisuje, kako se v programsko kodo vključi napako, se imenuje *mutacijski operator*. Več o mutacijskih operatorjih je opisano v poglavju 2.3.

Drugi korak obsega testiranje mutantov s testnimi primeri. Mutantu rečemo, da je *ubit*, če vsaj eden od testov zazna napako oziroma zazna, da se program ne obnaša tako kot bi se moral. V nasprotnem primeru rečemo, da je mutant *preživel* oziroma, da je ostal *živ*. Proces mutacijskega testiranja lahko tudi opišemo po naslednjih korakih:

1. Izhodišče je program P.
2. Na P apliciramo mutacijske operatorje in dobimo množico mutantov M.
3. Program P in vsakega mutanta *m* iz množice M ($m \in M$) testiramo s testom *t* iz množice testov T ($t \in T$).
4. Analiziramo rezultate testiranja za P in posameznega *m*.
5. Če so rezultati testiranja različni, je vsaj eden od testov *t* iz T zaznal napako in ubil mutanta *m*.
6. Če so rezultati testiranja enaki:
   - je mutanta *m* težko ubiti in je potrebno teste popraviti ali jih dopisati, da bodo zaznali napako ali
   - ima mutant *m* enak semantičen pomen kot program P oziroma je ekvivalenten.

Ekvivalentni mutanti so mutanti, ki se od originalnega programa razlikujejo sintaktično, semantično pa so enaki – izhod mutanta je vedno enak izhodu originalnega programa. Le-teh testi ne morejo ubiti in povzročajo odstopanje v rezultatu oziroma negativno vplivajo na rezultat analize, saj kažejo na šibkost testov, ki je v resnici lažna. Zaradi tega razloga je potrebno ekvivalentne mutante izločiti iz rezultatov. Prav tako jih samodejno ni mogoče odkriti, saj ne obstaja pravilo, ki bi enolično določalo ekvivalentnost mutantov. Odkrivanje ekvivalentnih mutantov mora uporabnik izvajati ročno, kar je eden bolj zamudnih delov mutacijskega testiranja in je del *problema izvedljivih poti* (angl. feasable path problem). [28]

Ekvivalentnih mutantov ne smemo zamešati s *trmastimi mutanti* (angl. stubborn mutants). Za razliko od ekvivalentnih mutantov trmaste mutante s testi lahko ubijemo, vendar so ti testi bolj specifični oziroma mutante lahko ubijemo z zelo omejenim naborom vhodnih parametrov, ki povzročijo različno obnašanje od obnašanja originalnega programa. Prav tako je mutant trmast le, če zanj še nismo odkrili testa, ki bi ga ubil. [16]

Mutacijska ocena (angl. mutation score ali krajše MS) je rezultat testa ali množice testov analize. Pove nam, kako dobro je določen test zaznal napake v programu in posledično koliko je test močan. Definirana je kot razmerje med številom ubitih mutantov in številom vseh neekvivalentnih mutantov, kar prikazuje enačba (2.1) na naslednji strani.

$$MS = \frac{\text{št. ubitih mutantov}}{\text{št. vseh neekvivalentnih mutantov}}, kjer\ je\ 0 \leq MS \leq 1\ ali\ 0\% \leq MS \leq 100\% \quad (2.1)$$

Nizka ocena pomeni, da testi v veliki meri ne zaznajo vstavljenih napak, visoka ocena pa pomeni, da so testi odkrili večino ali vse napake. Test je zaznal vse mutante in je mutacijsko ustrezen (angl. mutation adequate) ali idealen, če je njegova mutacijska ocena 100%.

2.2 Slabosti mutacijskega testiranja in možne rešitve

Mutacijsko testiranje se spopada z dvema večjima težavama. Prva težava je časovna in računska zahtevnost. Razlog je v tem, da že zelo majhen program lahko tvori veliko število mutantov, vsak mutant pa je potrebno testirati z vsemi testnimi primeri. Število generiranih mutantov za posamezen program je sorazmerno s produktom števila podatkovnih sklicev in števila podatkovnih objektov. [29]

Druga težava je negotovost pri določanju ekvivalentnih mutantov. Za reševanje prve težave so bile predstavljene številne tehnike, ki so razdeljene na tri strategije: "naredi manj" (angl. do fewer), "naredi pametneje" (angl. do smarter) in "naredi hitreje" (angl. do faster). [29] Omenjene strategije so opisane v naslednjih treh podpoglavjih, reševanje težave ekvivalentnih mutantov pa je opisano v poglavju 2.2.4.

2.2.1 Strategija "naredi manj"

Strategija "naredi manj" poskuša znižati število generiranih mutantov na način, da znižanje ne bi občutno vplivalo na učinkovitost testiranja.

Prvi pristop k strategiji je t.i. *selektivna mutacija* (angl. selective mutation), ki so ga prvi predstavili Jeff Offutt in njegova ekipa raziskovalcev. [29] Cilj je uporaba čim manj, vendar čim bolj učinkovitih mutacijskih operatorjev, saj imajo nekateri bistveno manjši učinek kot drugi. Izbira manjšega števila operatorjev povzroči manjše število generiranih mutantov, kar pomeni znižanje časovne zahtevnosti mutacijske analize.

Drugi pristop k strategiji, ki ga je prav tako razvila ekipa pod vodstvom Jeffa Offutta, je t.i. *mutacijsko vzorčenje* (angl. mutation sampling). [29] Tehnika mutacijskega vzorčenja naključno izbere podmnožico generiranih mutantov. Podmnožica je nato testirana z naborom testov, da se določi njena učinkovitost. Če podmnožica ne zadostuje zadanemu kriteriju se naključno izbere nova podmnožica mutantov. Podmnožice se tako izbirajo, dokler ne najdemo podmnožice, ki najbolj učinkovito določa učinkovitost določenega testa. Poleg iskanja podmnožice mutantov z ugibanjem pa obstaja tudi možnost uporabe verjetnostnih testov (npr. Bayesov test), ki statistično določijo primerni vzorec mutantov.

Eden od možnih pristopov je tudi generiranje *mutantov višjega reda*. Mutanti drugega reda vsebujejo dve napaki, kar dosežemo z združitvijo dveh mutantov prvega reda. Na ta način lahko število mutantov razpolovimo (če vsakega od mutantov prvega reda uporabimo le enkrat in ne upoštevamo vseh možnih kombinacij, zaradi katerih pride do ponavljanja). Poleg tega je pri mutantih, ki vsebujejo dve ali več napak, verjetnost ekvivalentnosti veliko manjša. [13]

2.2.2 Strategija "naredi pametneje"

Cilj strategije "naredi pametneje" je izogibanje izvajanja nepotrebnih delov programa, uvedba sprememb na že prevedenem programu in razporejanje bremena mutacijske analize na več naprav.

Prvi pristop se imenuje *šibka mutacija* (angl. weak mutation) in je tehnika približevanja, ki primerja notranje stanje mutanta in originalnega programa takoj po izvedbi mutiranega dela programa. Če je notranje stanje mutanta drugačno kot stanje originalnega programa je testni primer ubil mutanta, v nasprotnem primeru pa mutant preživi. Šibka mutacija se od navadne (močne) mutacije razlikuje v tem, da pri šibki mutaciji ni nujno, da se testi izvedejo do konca. Testi morajo namreč doseči le mutirano vrstico in prepoznati spremembo v stanju programa, ni pa potrebno preveriti in primerjati končnega izhoda mutiranega programa z izhodom originalnega programa. Empirični rezultati navajajo, da lahko z uporabo šibke mutacije prihranimo približno 50% časa pri izvajanju mutacijske analize z le majhno izgubo učinkovitosti. [19], [29]

Ostali "naredi pametneje" pristopi se usmerjajo tudi na vzporedno izvajanje mutacijske analize na več napravah in na algoritme, ki pametno shranjujejo stanja in segmente kode, katere lahko tako večkrat ponovno uporabimo. Vzporedno izvajanje analize je preprosto, saj so mutanti med seboj neodvisni in jih lahko testiramo posamično. [29]

2.2.3 Strategija "naredi hitreje"

Strategija "naredi hitreje" ponuja pristope, ki pohitrijo tako generiranje kot testiranje mutantov.

Glavni pristop vključuje uporabo metode *generiranja mutacijske sheme* (angl. Mutant Schema Generation). [18] Namesto generiranja številnih mutantov za posamezne dele programa, generiramo en sam "metamutant", ki predstavlja vse možne napake. Pred izvedbo testov se tako prevede le metamutant (poleg originalnega programa in testov), nakar se označi napaka, katero želimo testirati. Poleg tega pa se metamutant izvede v enakem programskem okolju kot originalni program z namenom uvedbe napake. [18] Primer metamutanta prikazuje slika 2.1. [35] Na ta način prihranimo čas pri generiranju in prevajanju mutantov. [21]

Originalni program	Metamutant

`int sestej (int a, int b)`
`  rezultat = a + b`
`  return rezultat`
	`switch (n)`
	`  case 1: rezultat = a - b`
	`  case 2: rezultat = a * b`
	`  case 3: rezultat = a / b`
	`  case 4: ...`

Slika 2.1: Primer metamutanta.

Nekateri pristopi za pohitritev temeljijo na takojšnem prevajanju izvorne kode mutantov, tako da se uporabi transformacija v *vmesno kodo* (angl. bytecode) ob nastanku mutantov. Na ta način prihranimo čas, ker se znebimo vsakokratnega sprotnega prevajanja mutantov, prav tako pa nam omogoča testiranje programov, katerih izvorna koda nam ni dostopna. [17], [27]

2.2.4 Reševanje problema ekvivalentnih mutantov

Članek [14] opisuje vpliv ekvivalentnih mutantov. V njem je dokazano, da so ekvivalentni mutanti pogosti. Z rezultati eksperimentov je potrjeno, da je ob izboru neprimernih mutacijskih operatorjev lahko tudi do 40% ustvarjenih mutantov ekvivalentnih. Vpliv mutacije na izvajanje programa lahko uporabimo za odkrivanje ekvivalentnih mutantov: manj kot mutacija spremeni izvajanje programa, večja je verjetnost, da je mutant ekvivalenten. Vpliv na izvajanje programa lahko določimo na primer s primerjavo toka nadzora (ang. control flow) pred in po mutaciji. Kot je opisano v članku, ekvivalentne mutante lahko razdelimo v štiri skupine:

- Prvi so ekvivalentni mutanti, ki vplivajo na neuporabljene dele kode in tako obnašanje programa ne spremenijo. Do takšnega primera lahko pride, če se določeno obnašanje podvaja na več mestih v kodi ali če se vrednost spremenljivk prepiše še pred njihovo uporabo.

- Sledijo ekvivalentni mutanti, ki zatrejo izboljšave v hitrosti. Primer je mutacija, ki povzroči, da se isti par ključ-vrednost večkrat vstavi v podatkovno strukturo slovar (angl. map), kar pa ne vpliva na samo delovanje programa.

- Tretja skupina zajema ekvivalentne mutante, ki spremenijo stanje zasebnosti razreda ali vrednost, ki jo vrne privatna metoda, a ne vplivajo na obnašanje programa.

- Nazadnje nastopajo še ekvivalentni mutanti, ki se ne izvedejo, ker se določeni pogoji ne izpolnejo. Vplivajo na del kode, ki je program v danem primeru ne doseže oziroma

ne uporabi in tako ne vplivajo na izhod programa. Primer je mutacija znotraj *if* bloka, pri čemer se *if* pogoj ne izpolne in se tako koda, ki jo vsebuje, ne izvede.

Za reševanje problema ekvivalentnih mutantov so bile razvite številne tehnike, ki implementirajo razne algoritme in so realizirane v obliki orodij za mutacijsko testiranje.

Prva takšna tehnika za odpravo ekvivalentnih mutantov uporablja posebno obliko *selektivne mutacije*, ki jo doseže z *genetičnimi algoritmi* in s *koevolucijo*. [2] V nasprotju z običajno selektivno mutacijo, tehnika ne zavrača mutacijskih operatorjev. Temelji na principu koevolucije – t.j. sprememba genetske sestave enega organizma, ki nastane kot odgovor na spremembo genetske sestave drugega organizma. Tehniko sestavljajo štirje koraki. V prvem koraku se ustvari nabor mutantov, ki jih program proizvede pri uporabi vseh mutacijskih operatorjev. Drugi korak zajema uporabo genetičnih algoritmov, ki razvijajo podmnožice mutantov in zavračajo nepomembne in neučinkovite mutante. Podobno kot v drugem koraku se v tretjem koraku izvaja genetična evolucija testnih primerov, pri čemer se vrednoti njihova učinkovitost glede na njihovo mutacijsko oceno. V zadnjem koraku se izvede koevolucija oziroma vzporedna evolucija mutantov in testnih primerov, pri čemer populacija mutantov tekmuje s populacijo testnih primerov. Koevolucija populacij določi mutante, katere je zelo težko ubiti, in testne primere, ki so sposobni ubiti zahtevne mutante. Tehnika zagotavlja, da ekvivalentni mutanti ne vplivajo na učinkovitost posameznih testov. To zagotovi z uporabo naprednih "fitnes" funkcij, ki zahtevajo, da vsakega mutanta ubije vsaj en test. [2]

Druga tehnika uporablja t.i. *rezanje programa* (angl. program slicing) za pomoč pri odkrivanju ekvivalentnih mutantov. [11] Rezanje programa oziroma deljenje programa na več delov lahko olajša proces odkrivanja ekvivalentnih mutantov, prav tako pa zmanjša potrebno delo, ki ga zahteva ročno iskanje ekvivalentnih programov. Rez mutiranega programa oziroma mutanta se navadno opravi na mestu, kjer je bila uvedena sprememba oziroma mutacija. Tako dobimo izsek mutanta, ki glede na programsko kodo sega od začetka do vpeljane napake. V program na predpisan način [11] uvedemo poljubno spremenljivko. Izsek kode nato testiramo, da ugotovimo, če je bil mutant ubit ali ne. Ekvivalentni mutanti naj bi vedno ustvarili izseke, ki so enaki za vse mutante. [11]

Tretja tehnika uporablja *optimizacijo prevajalnika*. [6], [10] Glavna misel avtorjev je, da so ekvivalentni mutanti lahko le optimizacije ali deoptimizacije originalnega programa. [6] Spremembe, ki jih uvedejo optimizatorji kode, so pravzaprav ekvivalentni mutanti. [10] Ekvivalentne mutante tako z algoritmi lahko vedno prepoznamo po njihovi optimizirani kodi v primerjavi z izvorno kodo programa.

Četrta tehnika domneva, da je problem iskanja ekvivalentnih mutantov v resnici neke vrste *problem izvedljivih poti* (angl. feasible path problem). [28] Problem izvedljivih poti trdi, da so nekatere testne zahteve nemogoče oziroma neizvedljive zaradi semantike programa. Tem zahtevam program preprosto ne more ustrezati, zato uporabimo tehniko *testiranja na*

osnovi omejitev (angl. constraint-based testing). Le-ta določi matematične lastnosti oziroma pogoje, ki omejijo vhodno območje programa na točno določeno vhodno domeno, ki zadostuje določenemu cilju in katerim morajo ustrezati testni primeri. Primer matematičnega pogoja je $x > 0$, ki opisuje del vhodnih podatkov, kjer je x pozitiven. Matematične pogoje nato uporabimo kot omejitev za programe. Omejitev predstavlja pogoje, pod katerimi je mutant ubit. Če test ubije mutanta, test zadostuje sistemu omejitev. Če sistem omejitev ne more biti izpolnjen, test za uboj mutanta ne obstaja in je tako mutant ekvivalenten. Omejitve na ta način določijo, kateri mutanti so ekvivalentni in kateri ne. Glavni pristop k uporabi omejitev za odkrivanje ekvivalentnih mutantov je iskanje neizvedljivosti v sistemu omejitev. [28]

2.3 Mutacijski operatorji

Mutacijski operator je nabor navodil za generiranje mutantov določene vrste. [5] Na primer, ko uporabimo *pogojni* mutacijski operator, se ustvari nova kopija izvorne kode programa, pri čemer se ena instanca dvočlenega pogojnega operatorja spremeni v drugega (primer je sprememba iz OR v AND). Mutacijski operatorji so podobni za vse vrste programskih jezikov, razlike nastanejo le pri lastnostih, ki so unikatne posameznim jezikom.

Njihov namen je oponašanje in reprodukcija tipičnih programerskih napak, prav tako pa so namenjeni uporabi značilnih testnih hevristik. Operatorji, ki na primer spremenijo sklice oziroma reference spremenljivk ali povzročijo zamenjavo relacijskih operatorjev, so primeri mutacijskih operatorjev, katerih namen je oponašanje tipičnih programerskih napak. Operator *failOnZero*() pa je primer slednjega namena saj spodbuja k uporabi testne hevristike "povzroči, da ima vsak izraz vrednost nič".

Pri izboru mutacijskih operatorjev, ki jih želimo uporabiti, se jih zdi smiselno vključiti čim več. Vendar veliko število mutacijskih operatorjev povzroči še večje število ustvarjenih mutantov, kar močno upočasni proces mutacijske analize. Zato pri analizi skušamo uporabiti manj mutantov, kar lahko dosežemo na dva načina: (1) naključno izbiranje mutantov iz množice vseh mutantov, ki jih ustvarijo mutacijski operatorji in (2) uporaba le najučinkovitejših mutacijskih operatorjev (učinkovitost mutacijskega operatorja se lahko spreminja glede na program, ki ga mutiramo).

Selektivna mutacija je izraz, ki opisuje strategijo uporabljanja le najučinkovitejših mutacijskih operatorjev. Pomen učinkovitosti je: če testi, ki so bili narejeni z namenom, da ubijejo mutante, ki jih je ustvaril mutacijski operator o_i, prav tako z visoko verjetnostjo ubijejo mutante, ki jih je ustvaril operator o_j, je mutacijski operator o_i bolj učinkovit kot o_j. Učinkoviti mutacijski operatorji vstavljajo enočlene operatorje (angl. unary operator) in spreminjajo tako enočlene kot dvočlene operatorje (angl. binary operator). Primer enočlenega

operatorja je *-a*, primer dvočlenega operatorja pa je *a - b*, pri čemer je v obeh primerih operator znak "-". [4]

Mutacijski operatorji so razvrščeni po jezikovnih konstruktih, ki jih spremenijo. Primera jezikovnih konstruktov sta metoda in razred, iz le-teh pa dobimo mutacijske operatorje, ki delujejo na nivoju metod (angl. method-level) in na nivoju razreda (angl. class-level). Sprva je bil obseg mutacijskih operatorjev omejen na nivo metod. [3] Operatorji, ki mutirajo izjave izključno znotraj metode se imenujejo *operatorji na nivoju metod* ali *tradicionalni operatorji*. Novejši operatorji so *razredno nivojski operatorji*, ki so namenjeni testiranju na nivoju razredov oziroma objektov. [24] Le-ti izkoriščajo prednosti objektno usmerjenih funkcij danega programskega jezika, prav tako pa so namenjeni širšemu spektru mutacij, ki obsegajo tako mutacije razreda kot mutacije med razredi. [34]

V naslednjih razdelkih so našteti in opisani **tradicionalni** mutacijski operatorji, kot sta jih v [4] definirala Paul Ammann in Jeff Offutt. Razredno nivojski mutacijski operatorji ne bodo opisani, saj jih področje naše raziskave ne zajema, njihove podrobnosti pa so prav tako na voljo v omenjeni knjigi.

2.3.1 ABS - Absolute Value Insertion

Mutacijski operator *vstavljanje absolutne vrednosti* preoblikuje vsakega od aritmetičnih izrazov in podizrazov s funkcijami *abs*(), *negAbs*() in *failOnZero*().

Funkcija *abs*() vrne absolutno vrednost vhodnega parametra oziroma izraza, funkcija *negAbs*() pa negacijo absolutne vrednosti. Funkcija *failOnZero*() preveri ali je vrednost izraza enaka nič. Če je vrednost izraza enaka nič, funkcija povzroči odpoved programa in je tako mutant ubit. V nasprotnem primeru funkcija vrne vrednost izraza in izvajanje programa se nadaljuje.

Uporabnik mora zagotoviti, da ima vsak številčen izraz vrednost nič, negativno vrednost in pozitivno vrednost, kar je tudi namen operatorja ABS. Uporaba operatorja na stavek "`x = 3 * a;`" proizvede naslednje tri mutirane stavke:

1. `x = 3 * abs(a);`
2. `x = 3 * -abs(a);` *oziroma* `x = 3 * negAbs(a);`
3. `x = 3 * failOnZero(a);`

2.3.2 AOR - Arithmetic Operator Replacement

Mutacijski operator *zamenjava aritmetičnih operatorjev* vsako pojavitev enega izmed aritmetičnih operatorjev `+`, `-`, `*`, `/`, `**` in `%` zamenja z vsakim od preostalih operatorjev. Poleg

tega vsako pojavitev aritmetičnega operatorja zamenjata tudi mutacijska operatorja *leftOp* in *rightOp*.

leftOp vrne levi operand (desni operand se ignorira), *rightOp* vrne desni operand, *mod* (`%`) pa izračuna ostanek pri deljenju levega operanda z desnim. Uporaba operatorja AOR na stavek "`x = a + b;`" proizvede naslednjih sedem mutiranih stavkov:

1. `x = a - b;`
2. `x = a * b;`
3. `x = a / b;`
4. `x = a ** b;`
5. `x = a % b;` *oziroma* `x = a mod b;`
6. `x = a;` *oziroma* `x = leftOp;`
7. `x = b;` *oziroma* `x = rightOp;`

2.3.3 ROR - Relational Operator Replacement

Mutacijski operator *zamenjava relacijskih operatorjev* vsako pojavitev enega izmed relacijskih operatorjev `<`, `≤`, `>`, `≥`, `=` in `≠` zamenja z vsakim od preostalih operatorjev in z mutacijskima operatorjema *falseOp* in *trueOp*.

falseOp vedno vrne vrednost *false*, *trueOp* pa vrednost *true*. Uporaba operatorja ROR na stavek "`if (m > n)`" proizvede naslednjih sedem mutiranih stavkov:

1. `if ( m >= n)`
2. `if (m < n)`
3. `if (m <= n)`
4. `if (m == n)`
5. `if (m != n)`
6. `if (false)` *oziroma* `if (falseOp)`
7. `if (true)` *oziroma* `if (trueOp)`

2.3.4 COR - Conditional Operator Replacement

Mutacijski operator *zamenjava pogojnih operatorjev* vsako pojavitev enega izmed pogojnih operatorjev `&&` (in), `||` (ali), `&` (kratkostični in), `|` (kratkostični ali) in `^` (ekskluzivni ali) zamenja z vsakim od preostalih operatorjev. Poleg tega vsako pojavitev pogojnega operatorja zamenjajo tudi mutacijski operatorji *falseOp*, *trueOp*, *leftOp* in *rightOp*.

leftOp vrne levi operand (desni operand se ignorira), *rightOp* vrne desni operand. *falseOp* vedno vrne vrednost *false*, *trueOp* pa vedno vrne vrednost *true*. Uporaba operatorja COR na stavek "`if (a && b)`" proizvede naslednjih osem mutiranih stavkov:

1. `if (a || b)`
2. `if (a & b)`
3. `if (a | b)`
4. `if (a ^ b)`
5. `if (false)` *oziroma* `if (falseOp)`
6. `if (true)` *oziroma* `if (trueOp)`
7. `if (a)` *oziroma* `if (leftOp)`
8. `if (b)` *oziroma* `if (rightOp)`

2.3.5 SOR - Shift Operator Replacement

Mutacijski operator *zamenjava pomikalnih operatorjev* vsako pojavitev enega izmed pomikalnih operatorjev `<<`, `>>` in `>>>` zamenja z vsakim od preostalih operatorjev in z mutacijskim operatorjem *leftOp*.

leftOp vrne nepomaknjen levi operand. Uporaba operatorja SOR na stavek "`x = m << a;`" proizvede naslednje tri mutirane stavke:

1. `x = m >> a;`
2. `x = m >>> a;`
3. `x = m;` *oziroma* `x = leftOp;`

2.3.6 LOR - Logical Operator Replacement

Mutacijski operator *zamenjava logičnih operatorjev* vsako pojavitev enega izmed bitnih logičnih operatorjev `&` (bitni in), `|` (bitni ali) in `^` (ekskluzivni ali) zamenja z vsakim od preostalih operatorjev in z mutacijskima operatorjema *leftOp* in *rightOp*.

leftOp vrne levi operand (desni operand se ignorira) in *rightOp* vrne desni operand. Uporaba operatorja LOR na stavek "`x = m & n;`" proizvede naslednje štiri mutirane stavke:

1. `x = m | n;`
2. `x = m ^ n;`
3. `x = m;` *oziroma* `x = leftOp;`
4. `x = n;` *oziroma* `x = rightOp;`

2.3.7 ASR - Assignment Operator Replacement

Mutacijski operator *zamenjava prireditvenih operatorjev* vsako pojavitev enega izmed prireditvenih operatorjev `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=` in `>>>=` zamenja z vsakim od preostalih operatorjev.

Uporaba operatorja ASR na stavek "`x += 3;`" proizvede naslednjih deset mutiranih stavkov:

1. `x -= 3;`
2. `x *= 3;`
3. `x /= 3;`
4. `x %= 3;`
5. `x &= 3;`
6. `x |= 3;`
7. `x ^= 3;`
8. `x <<= 3;`
9. `x >>= 3;`
10. `x >>>= 3;`

2.3.8 UOI - Unary Operator Insertion

Mutacijski operator *vstavitev enočlenih operatorjev* vstavi vsakega od enočlenih aritmetičnih (`+`, `-`), pogojnih (`!`) ali logičnih (`~`) operatorjev pred vsak izraz ustreznega tipa.

Uporaba operatorja UOI na stavek "`x = 3 * a;`" proizvede naslednje štiri mutirane stavke:

1. `x = 3 * +a;`
2. `x = 3 * -a;`
3. `x = +3 * a;`
4. `x = -3 * a;`

2.3.9 UOD - Unary Operator Deletion

Mutacijski operator *brisanje enočlenih operatorjev* izbriše vsakega od enočlenih aritmetičnih (`+`, `-`), pogojnih (`!`) ali logičnih (`~`) operatorjev pred vsakim izrazom ustreznega tipa.

Uporaba operatorja UOD na stavek "`if !(a > -b)`" proizvede naslednja dva mutirana stavka:

1. `if (a > -b)`
2. `if !(a > b)`

2.3.10 SVR - Scalar Variable Replacement

Mutacijski operator *zamenjava skalarnih spremenljivk* vsako pojavitev sklica na spremenljivko zamenja z vsako drugo spremenljivko primernega tipa, ki je deklarirana v trenutnem obsegu.

Operator SVR proizvede veliko število mutantov (V^2 mutantov, če je V število spremenljivk) in, ob uporabi prej navedenih mutacijskih operatorjev, ni nujen. Uporaba operatorja SVR na stavek "`x = a * b;`" proizvede naslednjih šest mutiranih stavkov:

1. `x = a * a;`
2. `a = a * b;`
3. `x = x * b;`
4. `x = a * x;`
5. `x = b * b;`
6. `b = a * b;`

2.3.11 BSR - Bomb Statement Replacement

Mutacijski operator *zamenjava stavka z bombo* vsak stavek programa zamenja s posebno funkcijo *Bomb*().

Ko se funkcija *Bomb*() izvede, povzroči odpoved programa. Tako morajo testi doseči vsak stavek, da lahko ubijejo vse mutante. Operator BSR proizvede en sam mutant na stavek programa in, ob uporabi mutacijskih operatorjev 1-9, ni nujen. Uporaba operatorja BSR na stavek "`x = a * b;`" proizvede naslednji mutiran stavek:

1. `Bomb();`

2.4 Primeri

Za boljšo predstavo in razumevanje smo postopek mutacijske analize ilustrirali s pomočjo treh preprostejših primerov.

Prvi primer vključuje uporabo mutacijskega operatorja AOR na osnovni aritmetični metodi, ki sešteje dve vhodni spremenljivki, pregled nastalih mutantov in postopek ubijanja mutantov s testi.

Drugi primer predstavi nekaj mutantov s pomočjo metode, ki vrne najmanjšega od dveh vhodnih parametrov.

Tretji primer pa predstavlja zgled ekvivalentnega mutanta, ki ga dobimo pri mutiranju metode, ki v zbirki vhodnih podatkov poišče največji element.

2.4.1 Primer z aritmetičnimi operatorji

Slika 2.2 vsebuje javansko metodo imenovano *Add* (levi stolpec slike), ki kot vhod prejme dve poljubni celi števili in kot izhod vrne njun seštevek. Torej predstavlja preprost primer aritmetične operacije seštevanja.

Za metodo *Add* smo napisali test z imenom *AddTest1* (slika 2.3, levi stolpec), ki bo testiral pravilno delovanje metode. Test od metode *Add*, ki kot vhod dobi števili 2 in 2, pričakuje izhodno vrednost 4. Metoda *Add* izvede izračun 2 + 2 in vrne vrednost 4, kot pričakuje test, torej metoda glede na test deluje pravilno. Vendar še vedno obstaja vprašanje "Ali je test dovolj dober, da nam zagotovi popolnoma pravilno delovanje metode?". Odgovor lahko dobimo s pomočjo mutacijske analize.

Originalna metoda	**Mutanti**

`int Add (int A, int B) {`
`  int C = A + B;`
`  return C;`
`} // end Add`
	`// 1. mutant`
	`int Add (int A, int B) {`
	`  int C = A - B;`
	`  return C;`
	`} // end Add`

	`// 2. mutant`
	`int Add (int A, int B) {`
	`  int C = A * B;`
	`  return C;`
	`} // end Add`

	`// 3. mutant`
	`int Add (int A, int B) {`
	`  int C = A / B;`
	`  return C;`
	`} // end Add`

	`// 4. mutant`
	`int Add (int A, int B) {`
	`  int C = A % B;`
	`  return C;`
	`} // end Add`

Slika 2.2: Metoda Add in njeni mutanti.

Metodo *Add* mutiramo z uvedbo mutacijskega operatorja AOR (*zamenjava aritmetičnih operatorjev*) na izvorno kodo metode in dobimo štiri mutante, ki so prikazani v desnem stolpcu slike 2.2. Za lažje razumevanje ne upoštevamo mutantov, ki ignorirata levi in desni operand. Vsak mutant predstavlja zamenjavo aritmetičnega operatorja seštevanja v originalni metodi z operatorji odštevanja, množenja, deljenja in modula. Mutante nato

posamezno testiramo z enakim testom (*AddTest1*) in primerjamo rezultate mutantov z rezultati originalne metode:

- Test prvega mutanta, ki kot vhod dobi števili 2 in 2, pričakuje izhodno vrednost 4. Prvi mutant izvede izračun 2 - 2 in vrne vrednost 0, kar test zazna kot napačen rezultat. Test je odkril napačno obnašanje in mutanta ubil.

- Test drugega mutanta, ki kot vhod dobi števili 2 in 2, pričakuje izhodno vrednost 4. Drugi mutant izvede izračun 2 * 2 in vrne vrednost 4. Mutant je vrnil pričakovano vrednost in glede na test deluje pravilno. Mutant ni pokazal napačnega obnašanja in ga zato test ni odkril. Drugi mutant je preživel, kar kaže na šibkost zastavljenega testa. Ker testa, ki bi ubil drugega mutanta, še nismo odkrili, je drugi mutant postal *trmasti mutant*.

- Test tretjega mutanta, ki kot vhod dobi števili 2 in 2, pričakuje izhodno vrednost 4. Tretji mutant izvede izračun 2 / 2 in vrne vrednost 1, kar test zazna kot napačen rezultat. Test je odkril napačno obnašanje in mutanta ubil.

- Test četrtega mutanta, ki kot vhod dobi števili 2 in 2, pričakuje izhodno vrednost 4. Četrti mutant izvede izračun 2 % 2 in vrne vrednost 0, kar test zazna kot napačen rezultat. Test je odkril napačno obnašanje in mutanta ubil.

Test *AddTest1* je ubil tri od štirih mutantov in si prislužil mutacijsko oceno 75%. Izkazalo se je, da test ni najbolj učinkovit, kljub temu, da je delovanje metode *Add* označil za pravilno. V primeru testa *AddTest1* se je operacija množenja obnašala enako kot operacija seštevanja zaradi neprimernega nabora vhodnih podatkov.

Test 1	**Test 2**

`@Test`
`void AddTest1 () {`
`  assertEquals (4, Add(2, 2));`
`} // end AddTest1`
	`@Test`
	`void AddTest2 () {`
	`  assertEquals (12, Add(8, 4));`
	`} // end AddTest2`

Slika 2.3: Testi za metodo Add.

Napisali bomo torej nov test *AddTest2* (slika 2.3, desni stolpec), ki bo metodi *Add* podal bolj naključne vhode. Nov test od metode *Add*, ki kot vhod dobi števili 8 in 4, pričakuje izhodno vrednost 12. Metoda *Add* izvede izračun 8 + 4 in vrne vrednost 12, kot pričakuje test,

torej metoda glede na test deluje pravilno. Isti test nato izvedemo še nad mutanti in primerjamo rezultate:

- Test prvega mutanta, ki kot vhod dobi števili 8 in 4, pričakuje izhodno vrednost 12. Prvi mutant izvede izračun 8 - 4 in vrne vrednost 4, kar test zazna kot napačen rezultat. Test je odkril napačno obnašanje in mutanta ubil.

- Test drugega mutanta, ki kot vhod dobi števili 8 in 4, pričakuje izhodno vrednost 4. Drugi mutant izvede izračun 8 * 4 in vrne vrednost 32, kar test zazna kot napačen rezultat. Test je odkril napačno obnašanje in mutanta ubil.

- Test tretjega mutanta, ki kot vhod dobi števili 8 in 4, pričakuje izhodno vrednost 4. Tretji mutant izvede izračun 8 / 4 in vrne vrednost 2, kar test zazna kot napačen rezultat. Test je odkril napačno obnašanje in mutanta ubil.

- Test četrtega mutanta, ki kot vhod dobi števili 8 in 4, pričakuje izhodno vrednost 4. Četrti mutant izvede izračun 8 % 4 in vrne vrednost 0, kar test zazna kot napačen rezultat. Test je odkril napačno obnašanje in je tako mutanta ubil.

Test *AddTest2* je ubil vse mutante in si prislužil mutacijsko oceno 100%. Prav tako je test *AddTest2* ubil tudi vse mutante, ki jih je ubil test *AddTest1*, kar pomeni, da je test *AddTest2* bolj učinkovit kot *AddTest1* in tako lahko test *AddTest1* popolnoma ovržemo. Z ubojem vseh mutantov smo tako ustvarili idealen test za metodo *Add*.

2.4.2 Primer na metodi Min

Slika 2.4 [4] vsebuje javansko metodo s šestimi mutiranimi vrsticami, ki so označene z grško črko delta (Δ). Vsak mutirani stavek predstavlja svoj ločen program, torej metodo *Min* z vnešeno spremembo delta. Vse mutacije so v danem primeru predstavljene znotraj ene metode zaradi boljše preglednosti.

Mutanti 1, 3 in 5 zamenjajo sklic na spremenljivko, mutant 2 spremeni relacijski operator in mutant 4 predstavlja posebno funkcijo "bomba", ki povzroči odpoved takoj, ko jo program doseže. Mutant 4 zahteva, da se izvede vsaka vrstica metode (ubijemo ga lahko le, če dosežemo "bombo", ki je v našem primeru zadnja vrstica metode) in tako zagotovimo pokritje kode. Mutant 6 ima prav tako posebno vlogo, saj povzroči odpoved, če je vhodni parameter enak nič, v nasprotnem primeru pa vrne vrednost vhodnega parametra oziroma ne vpliva na metodo. Ubijemo ga lahko samo, če ima B vrednost nič. Mutant 3 je primer ekvivalentnega mutanta, saj imata spremenljivki *minVal* in A na točki Δ3 enako vrednost in tako menjava ene spremenljivke za drugo na pravilno delovanje programa ne vpliva.

Ubijanje mutantov sledi modelu napak/odpovedi RIP (angl. RIP fault/failure model). [4] Dosegljivost (angl. **R**eachability), okužba (angl. **I**nfection) in razširitev (angl. **P**ropagation) se nanašajo na (1) doseganje mutanta, (2) spremembo stanja programa v nepravilno stanje, ki jo povzroči mutacija in (3) nepravilen izhod programa, ki je posledica nepravilnega stanja.

Kot primer vzemimo mutant 1. Mutant 1 se nahaja v prvi vrstici metode oziroma v prvem stavku programa, zato je pogoj dosegljivosti vedno zadoščen (*true*). Da lahko zagotovimo okužbo, mora biti vrednost spremenljivke B različna od vrednosti spremenljivke A, kar označimo z izrazom $A \neq B$. Pogoj razširitve izpolnemo, če mutirana različica metode *Min* vrne nepravilno vrednost. V našem primeru mora metoda *Min* vrniti vrednost, ki je bila prirejena v prvem stavku programa, kar pomeni, da se stavek $minVal = B$ znotraj bloka *if* **ne** sme izvršiti, saj bi to predstavljalo pravilno delovanje programa. Torej pogoj $B < A$ mora biti neresničen (*false*). Pogoji, ki jih mora test zagotoviti, da ubije mutanta, so:

- dosegljivost: *true*
- okužba: $A \neq B$
- razširitev: $(B < A) = false$

Test tako mutanta ubije, ko: $true \wedge (A \neq B) \wedge ((B < A) = \text{false})$

$$\equiv (A \neq B) \wedge (B \geq A)$$
$$\equiv (B > A)$$

Test z vhodom $A = 5$ in $B = 7$ ubije mutanta, saj bi originalna metoda vrnila rezultat 5 (A), mutirana različica pa vrne vrednost 7 (B).

Originalna metoda	**Metoda z vgrajenimi mutanti**

`int Min (int A, int B)`
`{`
`  int minVal;`
`  int minVal = A;`
`  if (B < A)`
`  {`
`  minVal = B`
`  }`
`  return minVal;`
`} // end Min`
	`int Min (int A, int B)`
	`{`
	`  int minVal;`
	`  int minVal = A;`
	Δ1 `  int minVal = B;`
	`  if (B < A)`
	Δ2 `  if (B > A)`
	Δ3 `  if (B < minVal)`
	`  {`
	`  minVal = B;`
	Δ4 `  Bomb();`
	Δ5 `  minVal = A;`
	Δ6 `  minVal = failOnZero(B);`
	`  }`
	`  return minVal;`
	`} // end Min`

Slika 2.4: Metoda Min in šest mutantov.

2.4.3 Primer ekvivalentnega mutanta

Slika 2.5 vsebuje javansko metodo in njen ekvivalenten mutant. Metoda *Max* kot vhod dobi seznam celih števil čez katerega se sprehodi s *for* zanko. Sproti si zapomni položaj oziroma indeks največjega števila in na koncu vrednost na tem indeksu vrne. Indeks največjega števila se spremeni samo v primeru, ko je število v dani iteraciji *for* zanke večje od števila na indeksu *indexMax*.

Na metodi *Max* (slika 2.5, zgoraj) uporabimo mutacijski operator ROR in dobimo nabor mutantov, pri čemer je mutant, ki je prikazan na sliki 2.5 spodaj, ekvivalenten. Od originalne metode se razlikuje po tem, da se indeks največjega števila spremeni samo v primeru, ko je število v dani iteraciji *for* zanke večje **ali enako** številu na indeksu *indexMax*.

Največje število je le eno, vendar se lahko pojavi večkrat, in sicer na različnih indeksih. Originalna metoda sproti primerja trenutno največje število z ostalimi števili v seznamu in indeks največjega števila spremeni le, če najde večje število. Mutirana metoda pa indeks največjega števila spremeni, če najde večje število **ali** če najde število z isto vrednostjo. Ker je pomembna samo vrednost največjega števila, indeks le-tega ne vpliva na izhod. Zato sta originalna metoda in njen mutant semantično enakovredna - torej je mutant ekvivalenten.

Semantično enakovrednost lahko dokažemo s testnim naborom vhodnih podatkov, kot prikazuje spodnji primer:

Vhodni podatki: `{2,5,1,5,3}`
Indeksi: `0,1,2,3,4`

Originalna metoda - spreminjanje spremenljivke indexMax tekom zanke *for*:

```
začetek:  indexMax = 0, vrednostMax = 2
1. korak: i = 1, indexMax = 1, vrednostMax = 5
2. korak: i = 2, indexMax = 1, vrednostMax = 5
3. korak: i = 3, indexMax = 1, vrednostMax = 5
4. korak: i = 4, indexMax = 1, vrednostMax = 5
```

Mutirana metoda - spreminjanje spremenljivke indexMax tekom zanke *for*:

```
začetek:  indexMax = 0, vrednostMax = 2
1. korak: i = 1, indexMax = 1, vrednostMax = 5
2. korak: i = 2, indexMax = 1, vrednostMax = 5
3. korak: i = 3, indexMax = 3, vrednostMax = 5
4. korak: i = 4, indexMax = 3, vrednostMax = 5
```

Iz primera lahko vidimo, da se kljub različnemu končnemu indeksu največjega števila vrednost le-tega ne spremeni. Iz tega lahko sklepamo, da ni nabora vhodnih podatkov, ki bi povzročil, da bi metoda vrnila napačen izhod, kar dokazuje, da je mutant ekvivalenten.

Originalna metoda	`int Max (int[] values) {`
	`  int indexMax = 0;`
	`  for (int i = 1; i < values.length; i++) {`
	`  if (values[indexMax] < values[i]) {`
	`  indexMax = i;`
	`  }`
	`  }`
	`  return values[indexMax];`
	`} // end Max`

Ekvivalenten mutant	`int Max (int[] values) {`
	`  int indexMax = 0;`
	`  for (int i = 1; i < values.length; i++) {`
	`  if (values[indexMax] <= values[i]) {`
	`  indexMax = i;`
	`  }`
	`  }`
	`  return values[indexMax];`
	`} // end Max`

Slika 2.5: Metoda Max in njen ekvivalenten mutant.

Poglavje 3 MuJava

µJava oziroma MuJava (daljše **Mu**tation System for **Java**) je mutacijski sistem za programe napisane v programskem jeziku Java. Nastal je kot produkt sodelovanja dveh univerz, Korea Advanced Institute of Science and Technology (krajše KAIST) v Južni Koreji in George Mason University v Združenih državah Amerike. Glavni raziskovalci in razvijalci sistema so Yu Seung Ma in prof. dr. Yong Rae Kwon iz Južne Koreje ter prof. dr. Jeff Offutt iz ZDA, velik doprinos pa je ustvaril tudi dr. Nan Li, ki je v veliki meri izboljšal delovanje sistema. MuJava je na voljo na spletni strani obeh univerz za raziskovalne in izobraževalne namene. [23]

Sistem MuJava omogoča generiranje, analizo in testiranje mutantov. Razvit je v programskem jeziku Java in je namenjen testiranju javanskih programov. Podpira tako generiranje mutantov na nivoju metod kot na razrednem nivoju in je predstavljen z grafičnim vmesnikom, ki pripomore k lažjemu izvajanju mutacijskega testiranja. Podpora generiranju in testiranju mutantov na razrednem nivoju zagotavlja mutacijsko testiranje objektno usmerjenih (angl. Object Oriented ali krajše OO) programskih jezikov in njihovih posebnih lastnosti kot so dedovanje, polimorfizem itd.. [22]

V splošnem mutacijska analiza podpira določanje testnega kriterija in ne testnega procesa. Kriterij je običajno vrednoten na podlagi raznih metrik pokritja. Mutacijski sistem je tako sestavljen iz dveh delov.

Prvi del upravlja mutacije na način, da izvorni program preda mutacijskemu generatorju, ki vsebuje zbirko mutacijskih operatorjev, le-ta pa generira številne variacije oziroma mutacije izvornega programa. Učinkovitost mutacijskega testiranja je zelo odvisna od izbranega nabora mutacijskih operatorjev, ki nadomestijo sintaktično ustrezne operatorje v izvornem programu.

Drugi del zajema testiranje novonastalih programov oziroma mutantov izvornega programa na način, da testne primere poda mutantom kot vhod. Testne primere najprej požene na izvornem programu in si zabeleži rezultate testiranja, nato pa iste testne primere izvrši nad vsemi mutanti, pri čemer si prav tako zabeleži rezultate. Rezultate izvornega programa nato primerja z rezultati posameznega mutanta. Neenakost rezultatov pove, da so testni primeri mutanta ubili. Iz tega lahko sklepamo, da so testni primeri dovolj učinkoviti, da zaznajo spremembo v izvornem programu. V idealnem primeru so testni primeri dovolj dobro napisani, da ubijejo vse mutante. Mutacijski sistem nam običajno prikaže tudi mutacijsko

oceno, ki je definirana kot razmerje med številom ubitih mutantov in številom vseh mutantov. [15]

MuJava implementira oba omenjena dela mutacijskega sistema, poleg tega pa zagotavlja tudi pregled nad mutanti. Pregled nad mutanti omogoča uporabniku prikaz programske kode mutantov, ki jih je ustvaril mutacijski generator. Podrobnosti in funkcionalnosti posameznih delov MuJave, so opisane v poglavju 3.2.

Velik tehnološki napredek na področju mutacijskih generatorjev predstavlja t.i. *generator shem* (angl. schemata generator). Le-ta ustvari mutacijsko shemo, ki predstavlja programsko okolico (angl. program neighborhood) in močno pospeši mutacijsko analizo. Mutacijska shema se sestoji iz dveh komponent, in sicer iz parametriziranega programa imenovanega *metamutant* in množice *metaprocedur*. Metamutant vključuje vse mutante izvornega programa v enem samem programu in jih hrani v parametrizirani obliki, pri čemer vsaka vrstica predstavlja spremembo v kodi oziroma mutacijo. Na ta način oblikujemo posamezne mutante tako, da iz metamutanta preprosto vzamemo spremembo oziroma mutacijo, ki jo želimo izvesti, in jo vstaviti v izvorni program. Generator mutantov MuJave je zasnovan na osnovi mutacijskih shem in metamutantov, kar zagotavlja hitrejše generiranje mutantov. Več o generatorju shem in o metamutantih je v člankih [18] in [21].

Pogosti problem tradicionalnih orodij za mutacijsko testiranje je pomanjkanje avtomatizacije glavnih delov procesa. Glavne operacije kot so vnašanje in poganjanje testnih primerov nad izvornim programom in mutanti ter pregled rezultatov testiranja so v breme uporabniku. Ker so časovno močno potratne, jih MuJava v veliki meri odpravi z avtomatizacijo. Za delovanje MuJava od uporabnika zahteva le izvorni program in program, v katerem so navedeni testni primeri za izvorni program. Preostanek operacij se dogaja v ozadju, pri čemer jih uporabnik le proži, dodaten vnos parametrov pa ni potreben.

3.1 Namestitev

Sistem MuJava v smislu namestitve ni zahteven, vendar za pravilno delovanje od uporabnika zahteva nastavitev lastnosti. Deluje tako na operacijskem sistemu Windows kot na sistemih Unix in Linux. Sistem za uporabo potrebuje Java Development Kit 1.6 (krajše JDK 1.6) ali novejšo različico le-tega.

Za začetek so potrebne tri datoteke: *mujava.jar*, *openjava.jar* in *mujava.config*, ki so dostopne na uradni strani MuJave [23]. V času izvajanja eksperimenta je bila uporabljena četrta različica MuJave (MuJava Version 4), za katero je v nadaljevanju opisan postopek namestitve.

Po končani namestitvi sta uporabniku na voljo dva grafična vmesnika. Prvi je namenjen generiranju mutantov, drugi pa izvajanju testov nad mutanti.

3.1.1 Namestitev okolja sistema MuJava

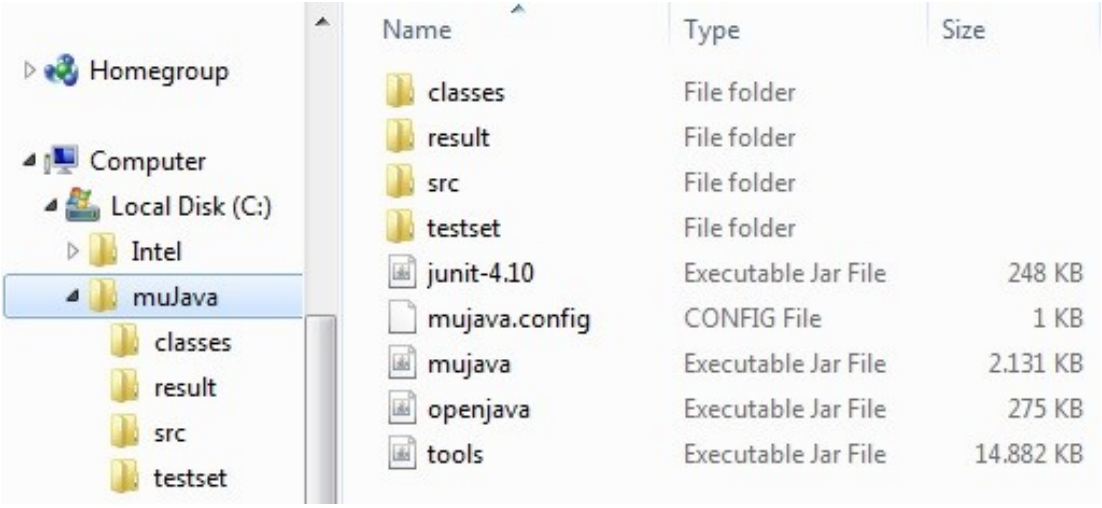
Prvi korak postavitve okolja sistema MuJava zahteva prenos treh datotek iz uradne spletne strani MuJave na trdi disk v imenik z dostopno potjo *MuJava_PATH* (npr. *MuJava_PATH = C:\muJava*). Sistemu mora biti zagotovljen tudi nabor javanskih orodij, ki so skupaj zbrana v datoteki *tools.jar* in so del Java Development Kita. Datoteka *tools.jar* se nahaja v imeniku *..\JDK\lib* in mora biti vključena v javansko okoljsko spremenljivko CLASSPATH. Za boljšo strukturiranost je bila datoteka *tools.jar* v našem primeru prepisana v imenik *MuJava_PATH* in se v spremenljivki CLASSPATH sklicuje od tam.

Naslednji korak zajema postavitev strukture imenikov. V imeniku *MuJava_PATH* ustvarimo imenike *src*, *classes*, *testset* in *result*. Imenike lahko ustvarimo ročno ali pa uporabimo razred *mujava.makeMuJavaStructure*, ki ga nudi *mujava.jar*. Razred pokličemo tako, da odpremo ukazno vrstico, se postavimo v imenik *MuJava_PATH* in uporabimo ukaz *java mujava.makeMuJavaStructure*. Namen posameznega imenika je opisan v tabeli 3.1.

Ime imenika	**Namen**
src	imenik za izvorne programe (končnica .java)
classes	imenik za prevedene razrede izvornih programov iz imenika *src* (končnica .class)
testset	imenik za testne primere oziroma prevedene javanske programe, ki implementirajo ogrodje jUnit (končnica .class)
result	imenik za generirane mutante (izhod programa za generiranje mutantov)

Tabela 3.1: Namen imenikov v strukturi sistema MuJava.

Testiranje mutantov prav tako zahteva knjižnico jUnit. Za naše potrebe pri testiranju je bila uporabljena jUnit različica 4.10, ki je na voljo na uradni spletni strani [20]. Datoteko vrste *.jar* prenesemo in shranimo v imenik *MuJava_PATH*. Na tej točki naj bi okolje v imeniku s potjo *MuJava_PATH* izgledalo, kot ga prikazuje slika 3.1.



Slika 3.1: Okolje sistema MuJava.

Zadnji korak zahteva nastavitev datoteke *mujava.config*. Datoteka *mujava.config* se mora nahajati v istem imeniku kot *mujava.jar* in *openjava.jar*. Služi kot kazalec na mesto oziroma imenik, kamor bo sistem MuJava shranjeval začasne datoteke. Ta kazalec lahko kaže na poljubno mesto, vendar je priporočljivo, da kazalec kaže na imenik s potjo *MuJava_PATH* ali njegov podimenik, ki je lahko tudi naknadno ustvarjen. Imenik mora biti opisan z absolutno potjo. Datoteko *mujava.config* odpremo s poljubnim urejevalnikom besedila in popravimo oziroma dodamo vrstico *MuJava_HOME=poljubnaAbsolutnaPot* (npr. *MuJava_HOME=C:\muJava* ali *MuJava_HOME=C:\muJava\tmp*).

3.1.2 Namestitev generatorja mutantov

Generator mutantov je že vključen znotraj datotek vrste *.jar*, ki so bile nameščene v prejšnjem poglavju. Obstajata dva načina za izvedbo programa: preko ukazne vrstice ali preko *batch* skripte. Sami smo se odločili za uporabo *batch* skripte, katere koda je prikazana na sliki 3.2. Uporaba skripte je za uporabnika bolj preprosta, saj je ukaze potrebno napisati le enkrat.

```
1  cd C:\muJava
2  set CLASSPATH=%CLASSPATH%;.;mujava.jar;openjava.jar;classes;tools.jar;
3  java mujava.gui.GenMutantsMain
```

Slika 3.2: Vsebina *batch* skripte za zagon programa za generiranje mutantov.

S prvim ukazom se postavimo v okolje sistema MuJava. Sistem MuJava zahteva vključitev *mujava.jar*, *openjava.jar* in javanski *tools.jar* v javansko okoljsko spremenljivko CLASSPATH. Prav tako v okoljsko spremenljivko dodamo imenik *classes*, saj generator

mutantov iz njega bere prevedene razrede izvornih programov. Tretji ukaz pokliče metodo za zagon uporabniškega vmesnika generatorja mutantov. *Batch* skripto ustvarimo tako, da v datoteko prepišemo vsebino slike 3.2 in jo shranimo kot *batch* datoteko oziroma s končnico *.bat*. Generator mutantov poženemo z dvoklikom na shranjeno datoteko.

Pred uporabo generatorja mutantov je potrebno predložiti izvorne programe, ki jih generator potrebuje za kreiranje mutantov, in jih postaviti v ustrezne imenike. Izvorne programe vrste *.java* postavimo v imenik *MuJava_PATH\src*, prevedene izvorne programe vrste *.class* pa v imenik *MuJava_PATH\classes*. Če je program vgnezden v paketno strukturo, postavimo celotno strukturo v ustrezen imenik. V primeru povezanih programov zaradi dedovanja, klicev ipd., v imenik postavimo tudi drugi program. Sistem MuJava ne preverja ali so vhodni programi prevedljivi ali ne, zato mora pravilnost delovanja zagotoviti uporabnik.

Po uspešnem generiranju mutantov generator shrani mutirane različice programa/programov v imenik *MuJava_PATH\result*. Shranijo se v podimenik z imenom, ki je enak izvornemu programu in so razdeljeni na tradicionalne in na razredne (angl. class) mutante.

3.1.3 Namestitev programa za testiranje mutantov

Podobno kot generator je tudi program za testiranje mutantov že vključen znotraj datotek vrste *.jar*, možno ga je izvršiti preko ukazne vrstice ali *batch* skripte in tudi zanj smo ustvarili skripto, katere vsebina je prikazana na sliki 3.3.

```
1  cd C:\muJava
2  set CLASSPATH=%CLASSPATH%;.;mujava.jar;openjava.jar;tools.jar;junit-4.10.jar;
3  java mujava.gui.RunTestMain > output.txt
```

Slika 3.3: Vsebina *batch* skripte za zagon programa za testiranje mutantov.

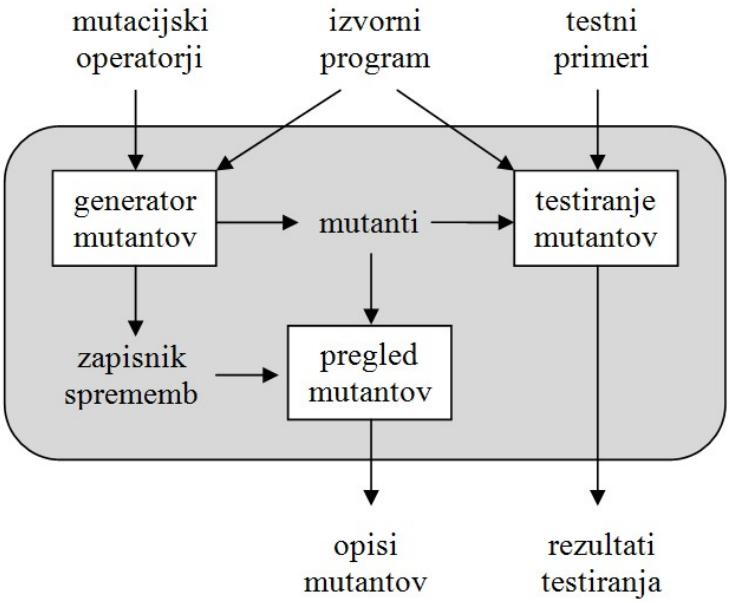
S prvim ukazom se postavimo v okolje sistema MuJava. Sistem MuJava zahteva vključitev *mujava.jar*, *openjava.jar* in javanski *tools.jar* v javansko okoljsko spremenljivko CLASSPATH. Prav tako v okoljsko spremenljivko dodamo knjižnico za ogrodje jUnit, s pomočjo katere bo lahko potekalo testiranje. Zelo pomembno je, da v okoljsko spremenljivko **ne** vključimo imenika *classes*, saj v nasprotnem primeru testiranje mutantov ne bo delovalo oziroma bodo rezultati napačni. Tretji ukaz pokliče metodo za zagon uporabniškega vmesnika programa za testiranje mutantov hkrati pa preusmerja izhod programa in ga zapisuje v datoteko *output.txt*.

Testni primeri v MuJavi morajo biti tipa jUnit. Vsak test mora biti metoda, ki vsebuje sekvenco klicev metod izvornega programa, ki ga testiramo. Testni razred in njegove metode

naj imajo javni dostop (angl. public access). Prevedene jUnit testne programe vrste *.class* postavimo v imenik *MuJava_PATH\testset*.

3.2 Funkcionalnosti

Sistem MuJava sestavljajo tri glavne komponente: generator mutantov, pregledovalnik mutantov in program za testiranje mutantov. Posplošena oblika strukture sistema in relacije med vhodi in izhodi sistema so predstavljeni na sliki 3.4 [22]. Vsaka izmed komponent ima tudi svoj uporabniški vmesnik.



Slika 3.4: Strukturna arhitektura sistema MuJava.

Sistem MuJava je avtomatiziran do stopnje, kjer je uporabniku potrebno le izbrati želene mutacijske operatorje in predložiti izvorni program ter teste za izvorni program. Kot rezultat sistem proizvede mutante vhodnega programa (če mutacije za izbrane operatorje obstajajo) in predstavi podatke o zaključenem testiranju mutantov: število ubitih mutantov, število živih mutantov in mutacijsko oceno.

3.2.1 Generator mutantov

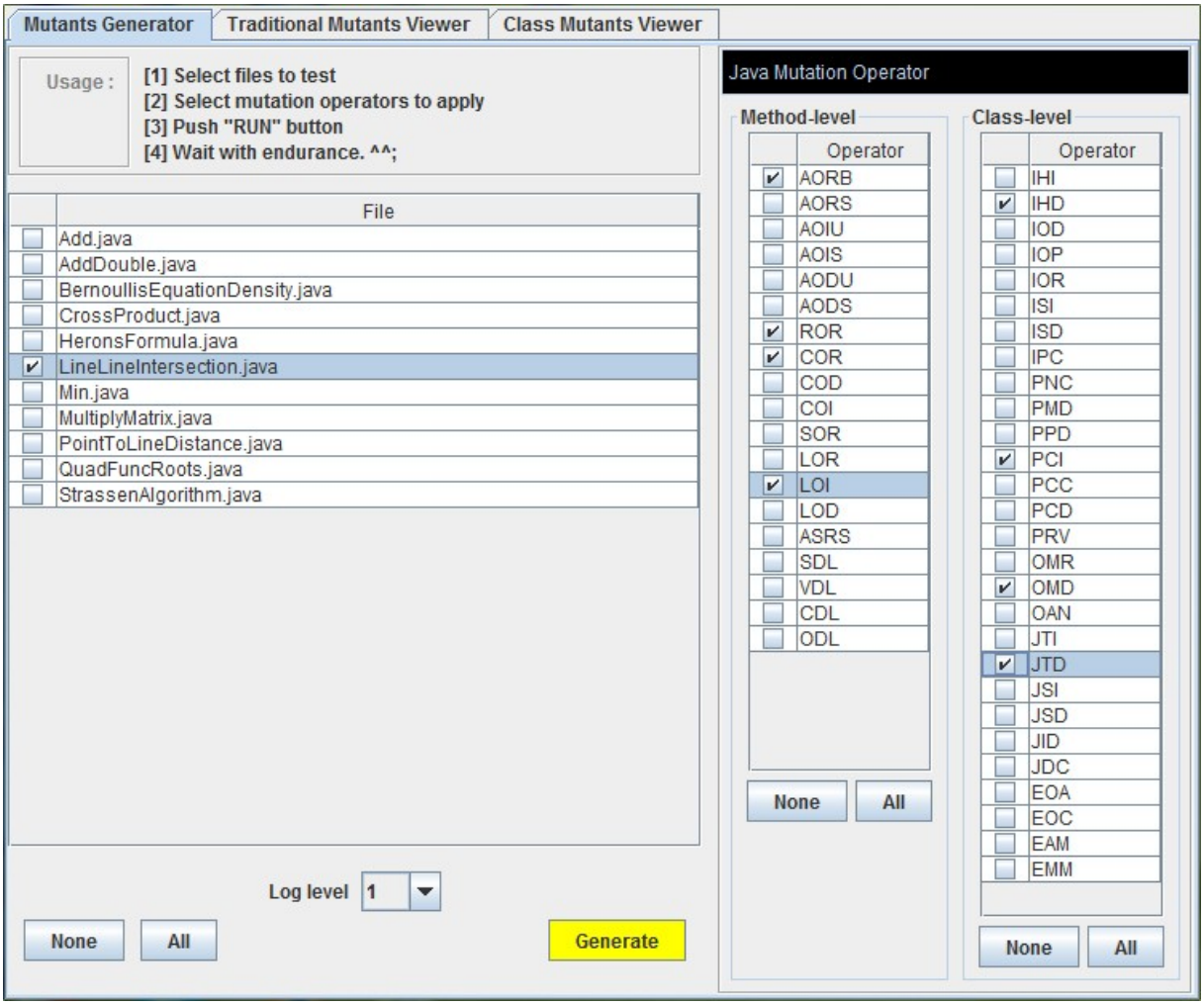
Generator mutantov je javanski program, ki generira tako tradicionalne kot razredne mutante. Med tradicionalne mutante sodijo mutacije, ki spremenijo operacije na nivoju metod in vplivajo na aritmetične, pogojne, logične in podobne izraze. Med razredne mutante sodijo mutacije, ki spremenijo operacije in relacije na razrednem nivoju in razkrivajo težave pri dedovanju, polimorfizmu, enkapsulaciji in drugimi lastnostmi objektov.

Mutanti se imenujejo po vrstah mutacijskih operatorjev, ki so bili uporabljeni na izvornem programu. V tabeli 3.2 je navedenih nekaj vzorcev, po katerih so ustvarjene MuJavine kratice in njihove razlage [34].

Nivo	**Vzorec (* = katerakoli črka)**	**Pomen**
metoda	AO**	aritmetičen izraz (+)
	ROR	relacija oz. enakost (==)
	CO*	pogojni izraz (!, &&)
	S**	pomik (<<)
	L**	logični izraz (&)
	ASRS	prirejanje (+=)
razred	I**	dedovanje (angl. **I**nheritance)
	P**	polimorfizem
	O**	preobremenjevanje (angl. **O**verloading)
	J**	za **J**avo značilne operacije
	E**	med-objektne operacije

Tabela 3.2: Mutacijski operatorji sistema MuJava.

Uporabnik lahko izbere programe za katere želi ustvariti mutante in mutacijske operatorje. Mutanti se najprej ustvarijo v obliki izvorne kode (angl. source code) nato pa se prevedejo v vmesno kodo (angl. bytecode) [22]. Po generiranju so informacije o posameznem mutantu prikazane v pregledovalniku mutantov. Izvorna koda generiranih mutantov je dostopna v imeniku *MuJava_PATH\result*. Slika 3.5 prikazuje grafični vmesnik generatorja mutantov.



Slika 3.5: Grafični vmesnik generatorja mutantov.

Grafični vmesnik je razdeljen na dva dela. Levi predel je namenjen izboru programa oziroma programov, za katere želimo generirati mutante, desni predel pa je namenjen izboru mutacijskih operatorjev, ki jih želimo uporabiti na izbranih programih.

Gumba "All" in "None" sta namenjena hitremu izboru vseh ali nobenih predmetov v določenem predelu.

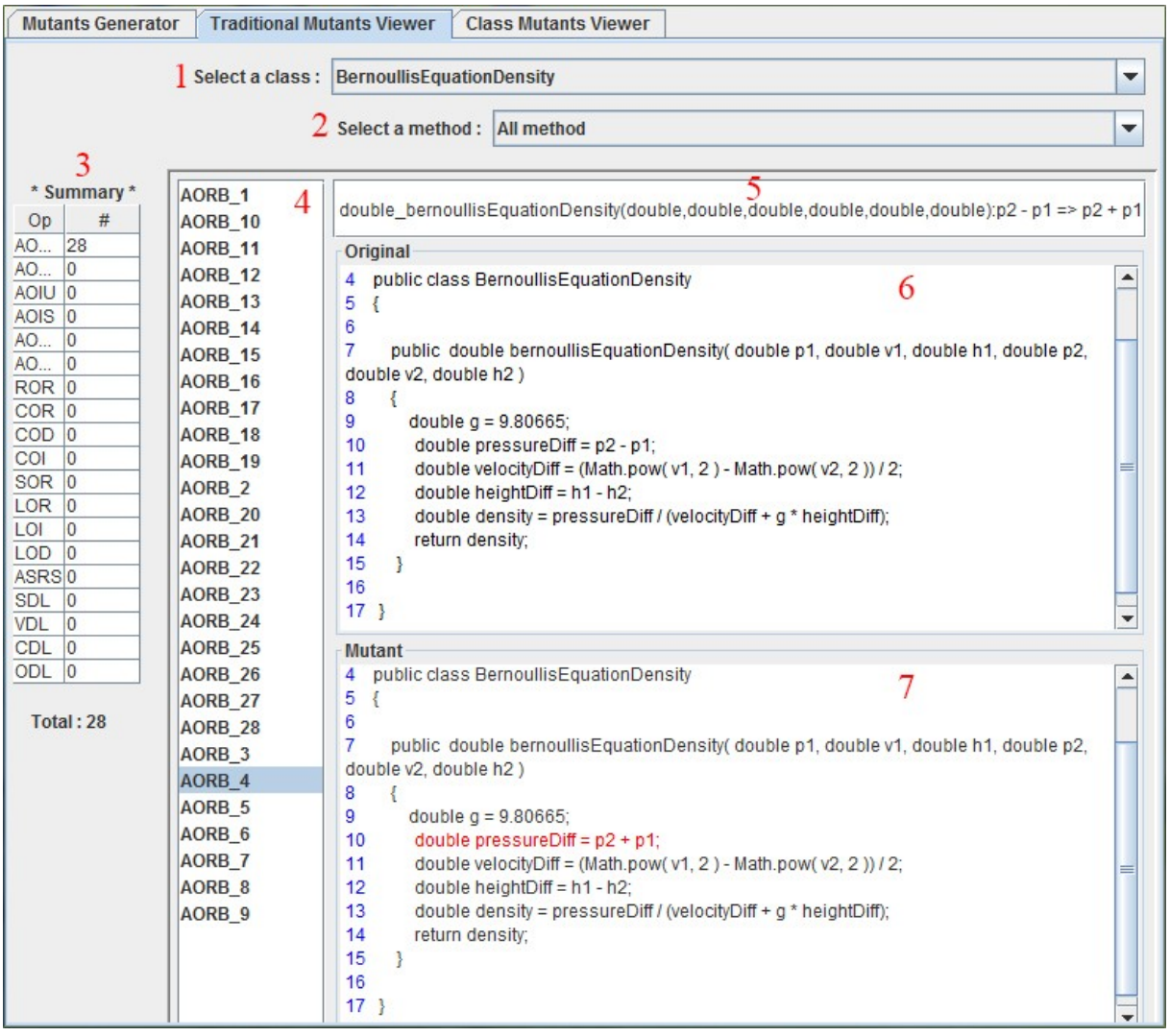
Grafični vmesnik ne vsebuje statusne vrstice, zato se vse informacije o statusu programa in napake beležijo v ukazni vrstici, ki je sočasno odprta z generatorjem mutantov. Stopnjo informiranja in beleženja v ukazno vrstico nastavimo z izbiro nivoja v spustnem seznamu "Log level".

Ko so želeni programi in mutacijski operatorji izbrani, gumb "Generate" začne proces generiranja mutantov. Proces generiranja lahko traja več časa, kar je odvisno od števila izbranih programov in mutacijskih operatorjev ter od zahtevnosti posameznih programov.

3.2.2 Pregledovalnik mutantov

Pregledovalnik mutantov je del tako programa za generiranje mutantov kot programa za testiranje mutantov in je ugnezden pod zavihka "Traditional mutants viewer" in "Class mutants viewer". Prvi zavihek prikaže mutante, ki so bili ustvarjeni na nivoju metod, drugi zavihek pa na razrednem nivoju. Prav tako prikazuje spremenjeni del izvorne kode za vsako posamezno mutacijo.

Pregled nad generiranimi mutanti je uporabniku v pomoč pri ustvarjanju dodatnih testnih primerov za mutante, ki jih je težko ubiti, in pri iskanju ekvivalentnih mutantov [22]. Grafični vmesnik pregledovalnika mutantov je prikazan na sliki 3.6. Posamezne komponente so oštevilčene za lažjo razlago njihovih funkcionalnosti.



Slika 3.6: Grafični vmesnik pregledovalnika mutantov.

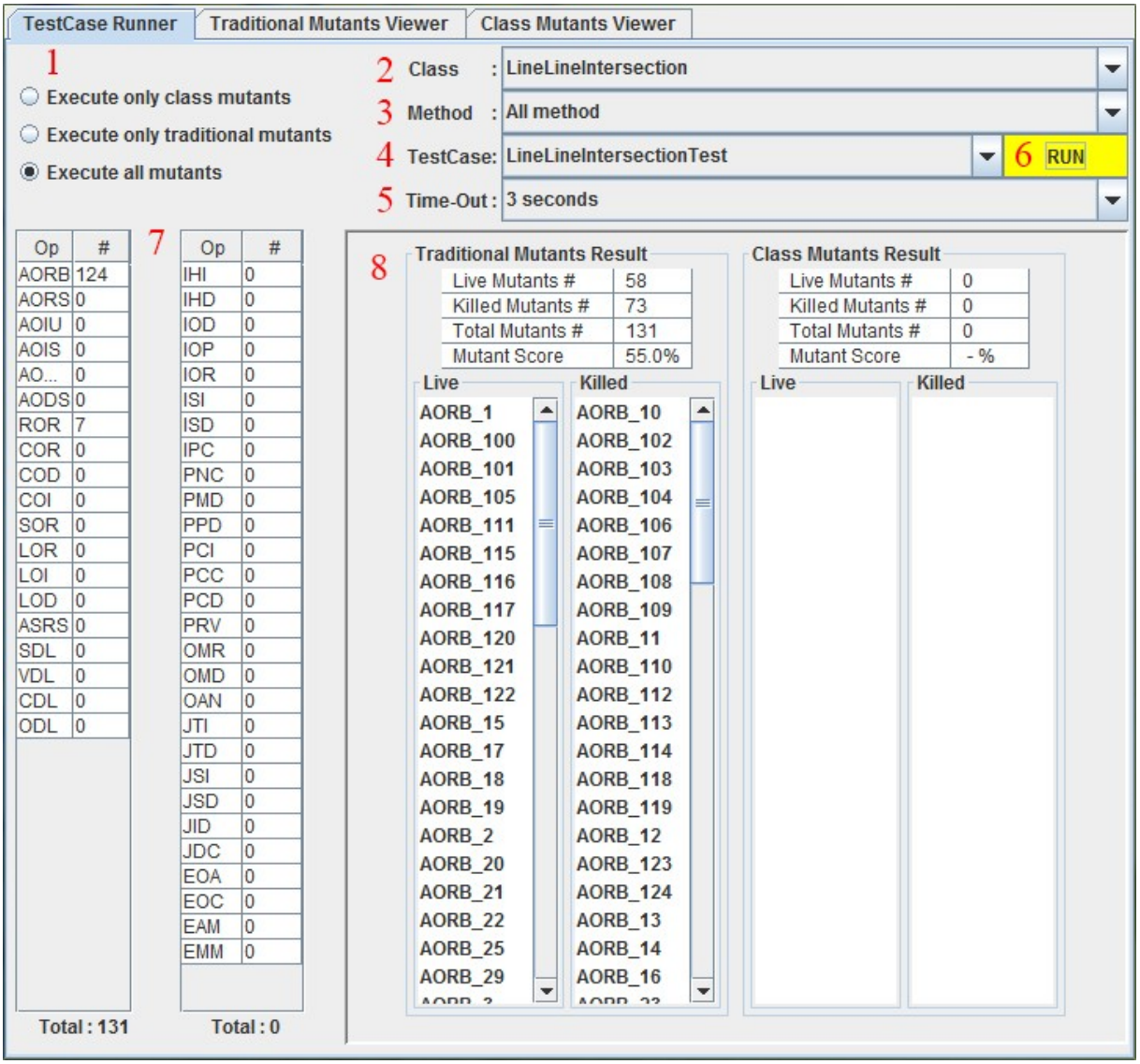
1. Spustni seznam za izbiro programa nad katerim želimo imeti pregled.
2. Spustni seznam, s katerim lahko omejimo prikaz na posamezne metode programa ali pa prikažemo mutante znotraj vseh metod.
3. Razpredelnica, ki prikazuje koliko mutantov je proizvedel posamezen mutacijski operator.
4. Seznam generiranih mutantov - izbor mutanta znotraj seznama prikaže njegovo izvorno kodo v predelu "Mutant".
5. Polje, kjer je zabeležena sprememba oziroma mutacija, ki je bila vpeljana pri izbranem mutantu (npr. sprememba minusa v plus).
6. Polje, kjer je prikazana izvorna koda originalnega programa.
7. Polje, kjer je prikazana izvorna koda mutiranega programa oziroma mutanta (vrstica, kjer je bila izvedena sprememba oziroma mutacija, je obarvana rdeče).

3.2.3 Program za testiranje mutantov

Program za testiranje mutantov preda mutante testnemu okolju (v našem primeru okolju jUnit), ki nad mutanti izvede enake teste kot nad originalnim programom in nato primerja rezultate posameznih mutantov z rezultati originalnega programa. Če je rezultat testiranja mutanta različen od rezultata originalnega programa, je mutant ubit. V nasprotnem primeru mutant preživi.

Testiramo lahko tako mutante na nivoju metod kot mutante na razrednem nivoju. Tako lahko izbiramo med testiranjem tradicionalnih mutantov (nivo metod), razrednih mutantov (razredni nivo) ali vseh mutantov.

Za začetek testiranja izberemo želeni program in jUnit teste ter pritisnemo gumb "RUN". Ko je testiranje končano, se prikaže število preživelih mutantov, število ubitih mutantov, mutacijska ocena in seznam ubitih ter seznam preživelih mutantov. Grafični vmesnik programa za testiranje mutantov je prikazan na sliki 3.7.



Slika 3.7: Grafični vmesnik programa za testiranje mutantov.

1. Izbira med testiranjem tradicionalnih mutantov (nivo metod), razrednih mutantov (razredni nivo) ali vseh mutantov.
2. Spustni seznam za izbiro programa katerega mutante želimo testirati.
3. Spustni seznam, s katerim lahko omejimo testiranje na posamezne metode programa ali pa testiramo celoten program.
4. Spustni seznam, s katerim izberemo teste.
5. Spustni seznam za izbiro najdaljšega časa, ki je dovoljen za testiranje posameznega mutanta.
6. Gumb, ki sproži začetek testiranja.
7. Razpredelnici, ki prikazujeta, koliko mutantov je proizvedel posamezni mutacijski operator (levo tradicionalni mutanti, desno razredni mutanti).

8. Razpredelnici s številom preživelih mutantov, številom ubitih mutantov, številom vseh mutantov, mutacijsko oceno, seznamom ubitih in seznamom preživelih mutantov (levo tradicionalni mutanti, desno razredni mutanti). V primeru, ko vrste (tradicionalni, razredni) mutantov ni, ostanejo polja za to vrsto prazna.

3.3 Druga orodja

Poleg sistema MuJava obstajajo tudi druga orodja za mutacijsko testiranje. V nadaljevanju so opisani sistemi, ki so bili potencialni kandidati za izvajanje našega eksperimenta. Sledi razlaga, zakaj je bil sistem MuJava primernejša izbira.

3.3.1 MuClipse

MuClipse je vtičnik (angl. plugin) za razvojno okolje Eclipse, ki ponuja povezavo med mutacijskim sistemom MuJava in okoljem Eclipse. Torej je drugačna izvedba orodja MuJava z enakimi funkcionalnostmi, vendar v obliki vtičnika za Eclipse.

V primerjavi z MuJava MuClipse zagotavlja posebne nastavitve znotraj sistema Eclipse za lažje izvajanje številnih meta-jezikovnih manipulacij, ki so značilne za mutacijsko testiranje, prav tako pa skrbi za pravilno razredno pot (angl. classpath). Ta ne sme vsebovati izvorne kode generiranih mutantov, katere želimo testirati in skušamo ubiti. Ponuja določeno nadgradnjo sistema MuJava na področju uporabnosti in združljivosti, saj lahko izvajamo mutacijsko testiranje kar v razvojnem okolju in ne v ločenem sistemu. Izboljšave vključujejo upravljanje razredne poti in strukture imenikov, nastavitve izvajanja programske kode in predstavitev rezultatov v integriranem grafičnem vmesniku. [33]

Vtičnik MuClipse je bil naša prva izbira, saj so bili vsi naši programi in njihovi testi napisani v razvojnem okolju Eclipse Luna. Tako bi lahko izvajali mutacijsko testiranje kar v razvojnem okolju s pomočjo vtičnika brez potrebe po migraciji programov v drugo okolje.

Žal se je podpora aktivnemu razvoju ustavila konec leta 2011 [32], kar je najverjetneje tudi razlog za našo neuspešno namestitev vtičnika v okolje Eclipse. Poskus namestitve preko vgrajenega "Eclipse Marketplace" namreč javi napako "java.lang.NullPointerException", kar nam prepreči tako namestitev kot uporabo vtičnika. Prav tako je bila zaradi varnostnih prijemov okolja Eclipse neuspešna ročna namestitev vtičnika, ki je na voljo tudi v obliki datoteke *.jar*. Zaradi nezmožnosti uporabe vtičnik MuClipse ni bil primeren kandidat za izvajanje našega eksperimenta.

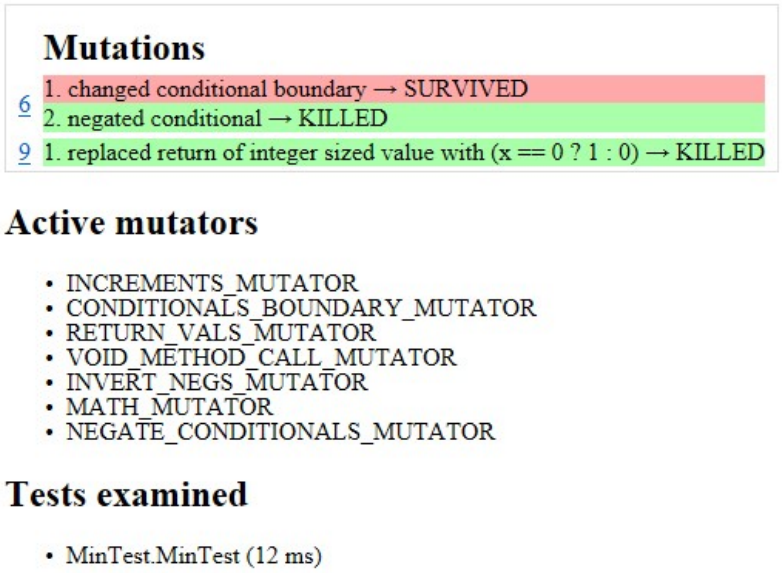
3.3.2 PIT Mutation Testing

PIT Mutation Testing ali krajše PIT je relativno novo orodje. Namenjeno je predvsem razvojnim ekipam, ki mutacijsko testiranje uporabljajo v praksi, in ne potrebam akademskih raziskav. Avtor meni, da je orodje PIT hitro, skalabilno in namenjeno realni programski opremi. [8]

Na voljo je kot samostojna aplikacija, ki jo je možno pognati preko ukazne vrstice ali kot vtičnik za razvojno okolje Eclipse. Ponuja deset mutacijskih operatorjev, od katerih jih je sedem privzeto vključenih. Mutacija se izvaja na prevedenih programih, podobno kot pri sistemu MuJava.

PIT vključuje lastno eksperimentalno funkcijo, ki se imenuje inkrementalna analiza (angl. incremental analysis). Temelji na (še nedokazani) predpostavki, da spremembe v odvisnostih znotraj razreda (angl. class depencencies) redko spremenijo rezultat mutacijskega testiranja. Inkrementalna analiza lahko sledi spremembam v osnovi programske kode in se na podlagi pridobljene informacije odloči, katere metode zahtevajo ponovno spremembo v naslednji izvedbi mutacijskega testiranja. Na ta način omogoča ponovno uporabo rezultatov prejšnjih analiz in posledično nemutiranih delov kode ni potrebno ponovno testirati (pod pogojem, da testi niso bili spremenjeni). Tako v veliki meri zmanjša čas izvajanja mutacijskega testiranja pri zaporednih izvedbah testov. [9]

Orodje PIT smo uporabili kot vtičnik (različica 1.0.0) za okolje Eclipse Luna. Mutacijsko testiranje je orodje izvajalo hitro in na uporabniku prijazen način. Testiranje namreč poženemo na enak način kot bi pognali testiranje jUnit.



Slika 3.8: Izpis rezultatov orodja PIT.

Rezultati se prikažejo v formatu *html* in opisujejo, kakšna je pokritost kode, koliko mutantov je bilo ubitih glede na vse mutante, katere vrste mutacij so bile uporabljene in z besedami opisujejo, kaj je mutacija povzročila v posamezni vrstici, kar prikazuje slika 3.8.

Prvi problem nam je predstavljala premajhna transparentnost orodja. Iz besednega opisa ni bilo mogoče brez ugibanja ugotoviti, kakšno spremembo je opravila posamezna mutacija.

To je razvidno iz slike 3.8; pod mutacijami ima v šesti vrstici prva mutacija opis "1. changed conditional boundary --> SURVIVED". Šesto vrstico programa predstavlja koda *if (b < a)*. Iz opisa rezultata lahko izvemo, da je mutacija vplivala na pogoj, ki preverja enakost spremenljivk a in b, vendar ne izvemo, kakšna je bila ta mutacija, ali je b manjše kot a (b < a) spremenila v b večje kot a (b > a) ali v b manjše ali enako a (b <= a) itd..

Drugi problem je premajhna fleksibilnost orodja. Orodje ima privzeto vklopljeno sedem mutacijskih operatorjev, katerih ne moremo prosto izbirati. Le-to nam ne da možnosti testiranja programov s poljubnim mutacijskim operatorjem, ampak moramo program testirati z vsaj sedmimi operatorji in iz rezultatov razbrati podatke, ki nas zanimajo. Prav tako mutacijski operatorji nimajo tradicionalnih imen, zato je težje razumeti kakšno mutacijo izvede kateri operator.

Naša raziskava je potrebovala boljši nadzor nad mutacijskimi operatorji in podrobnejšo preglednost nad učinki mutacij, zato orodje PIT, učinkovitosti in preprostosti uporabe navkljub ni bil primeren.

3.3.3 Judy, Jumble in Jester

Orodja za mutacijsko testiranje Judy [25], Jumble [30] in Jester [26] so bili prav tako možne alternative sistemu MuJava, vendar nobeden od naštetih ni ustrezal našim kriterijem.

Največja pomanjkljivost je prosta izbira mutacijskih operatorjev, ki je ne zagotavlja nobeno od naštetih orodij. Pregled nad generiranimi mutanti je majhen ali pa ga sploh ni. Mutacije ne obsegajo vseh možnih scenarijev (npr. Jumble [30]) in uporabniku neposredno niso vidne. Vsa našteta orodja se poganjajo preko ukazne vrstice, kamor tudi izpisujejo rezultate. Za testiranje mutantov vsa orodja uporabljajo knjižnico jUnit. [25]

Najprimernejši kandidat je bilo orodje Judy, saj kljub naštetim pomankljivostim podpira mutacije višjega reda (angl. Higher Order Mutations), predstavi in beleži bolj podrobne rezultate in je za razliko od orodij Jumble in Jester še vedno aktivno podprt. [25] Možnost izvajanja mutacij višjega reda bi nam sicer pohitrila izvajanje eksperimenta pri mutacijah drugega reda, vendar zaradi nezmožnosti izbiranja mutacijskih operatorjev izgubimo vso časovno prednost.

Poglavje 4 Analiza učinkovitosti mutacij mutacijskega operatorja AOR

4.1 Opis problema

Mutacijsko testiranje je močan in zelo učinkovit pristop k testiranju programske opreme, ki svojo pozornost usmerja na ustvarjanje kakovostnih testnih primerov za odkrivanje napak in nepravilnosti v delovanju programov. Vendar nič ni brez svojih slabih lastnosti in enako velja tudi za mutacijsko testiranje. Spopada se namreč s problemom časovne zahtevnosti, ki že pri manjših programih hitro uide izpod nadzora. Predstavljene so bile številne rešitve, ki strmijo k odpravi omenjenega problema in so opisane v poglavju 2.2.

Naš cilj je bil odkriti možne rešitve pri optimizaciji mutacijske analize na aritmetičnih programih, ki za generiranje mutantov uporablja mutacijski operator AOR oziroma *zamenjavo aritmetičnih operatorjev*.

Predpostavljali smo, da je smiselno število mutantov omejeno in je zato zamenjava vsakega aritmetičnega operatorja z vsemi ostalimi možnimi aritmetičnimi operatorji odvečno, potratno opravilo. Iz tega lahko sklepamo, da so določene mutacije aritmetičnih operatorjev bolj učinkovite pri odkrivanju napak kot druge. Mutacija je v našem primeru zamenjava danega aritmetičnega operatorja za drugega (npr. sprememba seštevanja v odštevanje) po postopku, kot ga opisuje poglavje 2.3.2.

Želeli smo ovrednotiti težo določene mutacije glede na to, ali je mutacija bolj vzdržna pri testih (oziroma jo je težje ubiti - večja teža) ali pa je neodporna in jo ubijejo že najosnovnejši testi (manjša teža). Za testiranje bi si želeli čimveč mutantov, katerih sprememba (mutacija) aritmetičnega operatorja predstavlja večjo težo, saj predpostavljamo, da mutacije z večjo težo pokrivajo tudi mutacije z manjšo težo in so zato slednje odvečne. Torej testi, ki ubijejo mutante večje teže, najverjetneje ubijejo tudi mutante z manjšo težo. Tako lahko zmanjšamo nabor aritmetičnih operatorjev, ki jih uporablja mutacijski operator AOR za zamenjavo aritmetičnih operatorjev v programu.

Primer: Z ovrednotenjem mutacij s pomočjo testiranja mutantov številnih programov smo ugotovili, da je mutante, ki so imeli poljuben aritmetičen operator spremenjen oziroma mutiran v množenje, seštevanje in deljenje, težje ubiti (večja teža) kot mutante, ki so imeli poljuben aritmetičen operator spremenjen v ostanek pri deljenju oziroma modul in odštevanje (manjša teža). Iz tega sklepamo, da pri nadaljnjem testiranju lahko uporabljamo le mutacije,

ki spremenijo poljuben aritmetičen operator v množenje, seštevanje ali deljenje, modul in odštevanje pa opustimo, saj nam prinašata zelo majhno dodano vrednost k rezultatom mutacijske analize. Na ta način bomo lahko v prihodnjih analizah generirali do 40% manj mutantov in s tem zmanjšali časovno zahtevnost le-teh.

4.2 Izvedba

Kot testni vzorec smo izbrali nabor osmih programov, ki predstavljajo rešitve preprostih aritmetičnih problemov. Ustreznost posamezne matematične enačbe določa število aritmetičnih operatorjev, ki jih enačba vsebuje: več operatorjev zagotovi generiranje večjega števila mutantov in posledično bolj natančno analizo dobljenih rezultatov.

Za realizacijo programov smo uporabili programski jezik Java in različico 4.4.0 razvojnega okolja Eclipse Luna. Posamezno enačbo smo implementirali v obliki metode, ki kot vhod prejme vrednosti spremenljivk in kot izhod vrne izračun enačbe. Ker sistem Mujava za izvajanje analize sprejema razrede in ne metode, smo posamezno metodo enkapsulirali v svoj razred, ki se imenuje enako kot metoda.

Za enačbe oziroma metode smo nato v ogrodju JUnit (različica 4.11) napisali ustrezne teste, ki testirajo pravilno delovanje posamezne metode tako, da metodi podajo vnaprej določene argumente in pričakujejo točno določen rezultat. Testi so namenoma napisani skromno in so predvsem namenjeni zagotovitvi stoodstotne pokritosti programske kode. V veliki večini testi obravnavajo le enega od možnih scenarijev oziroma en sam nabor vhodnih podatkov. Želeli smo namreč ponazoriti šibke teste, s katerimi bi lažje ovrednotili težo posameznih mutacij.

4.2.1 Potek eksperimenta z mutacijami prvega reda

Ustvarjene programe in njihove teste smo posredovali sistemu za mutacijsko testiranje. Za namen generiranja in testiranja mutantov smo uporabili sistem MuJava (različica 4). Najprej smo generirali mutante za posamezen program, pri čemer smo uporabili mutacijski operator AORB (črka B v AORB pomeni binary), ki vpliva le na aritmetične izraze. Dobljene mutante smo nato kategorizirali glede na učinek, ki ga je imela mutacija na originalen program, in tako dobili naslednjih pet kategorij:

1. Mutanti, ki so nastali pri mutaciji poljubnega aritmetičnega operatorja v plus,
2. mutanti, ki so nastali pri mutaciji poljubnega aritmetičnega operatorja v minus,
3. mutanti, ki so nastali pri mutaciji poljubnega aritmetičnega operatorja v množenje,
4. mutanti, ki so nastali pri mutaciji poljubnega aritmetičnega operatorja v deljenje in
5. mutanti, ki so nastali pri mutaciji poljubnega aritmetičnega operatorja v modul.

Namen razdelitve v kategorije je izolirano testiranje mutantov posamezne vrste aritmetične mutacije, s katerim bomo lahko ovrednotili težo le-te.

S sistemom MuJava smo nato testirali generirane mutante za vsak program posebej tako po posameznih kategorijah kot po kombinacijah med kategorijami. Beležili smo število ubitih mutantov, število preživelih mutantov, število vseh mutantov in mutacijsko oceno. Želeli smo ugotoviti, kakšno težo ima posamezna kategorija oziroma mutacija in kako se teža spreminja, če mutacije med seboj kombiniramo.

Kombinacije smo razdelili po stopnjah pri čemer prva stopnja opisuje rezultate vsake kategorije posebej, druga stopnja opisuje rezultate vseh kombinacij parov mutacij in peta stopnja opisuje rezultate vseh mutacij skupaj oziroma rezultat, ki bi ga sicer dobili, če bi testirali vse generirane mutante brez postopka kategorizacije. Prav tako smo vodili evidenco povprečne mutacijske ocene kategorij in njihovih kombinacij po stopnjah.

Po opravljenem testiranju mutantov vseh programov smo rezultate združili v eno samo razpredelnico in na podlagi vzorca ovrednotili težo posamezne mutacije, kot opisuje poglavje 4.4.

4.2.2 Potek eksperimenta z mutacijami drugega reda

Kar smo do sedaj testirali so bile mutacije prvega reda oziroma mutacije, ki v program vstavijo eno napako naenkrat in za vsako vstavljeno napako ustvarijo enega mutanta. Ovrednotenje mutacij pa smo želeli izvesti tudi na mutacijah višjega reda. Testiranje smo zato ponovno izvedli na mutacijah drugega reda oziroma na mutacijah, ki v program vstavijo dve napaki naenkrat, pri čemer obe napaki predstavljata enega mutanta.

Za generiranje mutantov drugega reda smo uporabili mutante prvega reda, ki smo jih ustvarili v prejšnjem eksperimentu. S sistemom MuJava smo na vsakem mutantu prvega reda uporabili mutacijski operator AORB in tako dobili mutante drugega reda. Pri tem smo upoštevali dejstvo, da tovrstno mutiranje ustvari določene neveljavne mutante, ki smo jih zato tudi izločili. Neveljavne mutacije so:

- mutacije drugega reda, ki ustvarijo že obstoječe mutante prvega reda in
- mutacije drugega reda, ki iz mutanta prvega reda tvorijo originalen program oziroma izničijo mutacijo.

Če mutiramo že spremenjeni operator mutanta prvega reda, kot rezultat dobimo drug operator prav tako mutanta prvega reda, saj nismo vpeljali dodatne napake, ampak le nadomestili prvo, kar prikazuje slika 4.1. Poseben primer je mutiranje že spremenjenega operatorja mutanta prvega reda v izvorni operator, pri čemer je mutant drugega reda enak originalnemu programu, kar je razvidno iz slike 4.2.

Neveljavna mutacija	**Originalni program**	**Mutant prvega reda**		**Mutant prvega reda**	**Mutant drugega reda**
1. operator	+ →	-	⇒	- →	*
2. operator	/	/		/	/

Veljavna mutacija	**Originalni program**	**Mutant prvega reda**		**Mutant prvega reda**	**Mutant drugega reda**
1. operator	+ →	-	⇒	-	-
2. operator	/	/		/ →	*

Slika 4.1: Primer veljavne in neveljavne mutacije drugega reda.

Neveljavna mutacija	**Originalni program**	**Mutant prvega reda**		**Mutant prvega reda**	**Mutant drugega reda**
1. operator	+ →	-	⇒	- →	+
2. operator	/	/		/	/

Slika 4.2: Mutacija mutiranega operatorja prvega reda nazaj v originalni program.

Število mutantov je močno naraslo. Podobno kot pri mutantih prvega reda smo tudi mutante drugega reda kategorizirali glede na učinek, ki ga je imela mutacija na originalen program. Ustvarili smo dvo-nivojsko kategorizacijo: prvi nivo predstavlja prva mutacija oziroma mutacija prvega reda, drugi nivo pa mutacija drugega reda. Tako smo dobili 25 kategorij oziroma 15 kategorij, če upoštevamo, da se kombinacije ponavljajo (mutacija +- je enaka kot mutacija -+).

Na primer prvih 5 kategorij od 25 predstavlja mutacijo prvega reda, ki spremeni poljubni aritmetični operator v plus in mutacije drugega reda, ki spremenijo poljubni aritmetični operator v plus, minus, množenje, deljenje ali modul.

Mutante smo nato testirali na podoben način kot mutante prvega reda, vendar nas tokrat kombinacije kategorij niso zanimale. Testirali smo generirane mutante po kategorijah za vsak program posebej in pri tem beležili število ubitih mutantov, število preživelih mutantov, število vseh mutantov in mutacijsko oceno.

Rezultate mutacijske analize vseh programov smo nato združili v eno samo razpredelnico in na podlagi vzorca ovrednotili težo posamezne mutacije drugega reda, kar opisuje poglavje 4.5.

4.3 Izbrani programi in njihovi testi

V naslednjih poglavjih so predstavljeni programi in njihovi testi, ki smo jih uporabili pri izvajanju eksperimenta. Programi so aritmetične narave in predstavljajo rešitve različnih matematičnih problemov.

4.3.1 Bernoullijeva enačba

Bernoullijeva enačba opisuje stacionarni laminarni tok nestisljive in neviskozne tekočine vzdolž tokovnice. [7] V primeru, ko je cev po kateri teče tekočina nagnjena in ima dva različna preseka, gostoto tekočine izračunamo po naslednji obliki Bernoullijeve enačbe (4.1):

$$\rho = \frac{p_2 - p_1}{\frac{1}{2} * (v_1^2 - v_2^2) + g * (h_1 - h_2)} \tag{4.1}$$

Spremenljivka *g* predstavlja gravitacijski pospešek, *p* predstavlja tlak, *v* hitrost, *h* višino in *ρ* gostoto.

Javanska metoda, ki implementira Bernoullijevo enačbo, njeni testi in rezultati mutacijskega testiranja prvega in drugega reda mutantov so predstavljeni v *prilogi 1*. Metodo smo testirali tako, da smo vrednost spremenljivk v_1 in h_1 nastavili na 1,0, medtem ko so spremenljivke v_2, h_2, p_1, p_2 nosile vrednost 0,0, pri čemer smo pričakovali, da bo metoda izračunala, da je vrednost gostote *ρ* enaka 0,0.

Mutacija metode je proizvedla skupno 28 mutantov prvega reda in 403 mutantov drugega reda. Mutacijska ocena testiranja vseh mutantov prvega reda je bila 21,4%, ocena testiranja mutantov drugega reda pa 43,2%.

4.3.2 Vektorski produkt

Vektorski produkt je binarna operacija nad dvema vektorjema v trirazsežnem prostoru. Vektorski produkt (4.2) vektorja *a* in *b* je trirazsežni vektor, ki je pravokoten na oba vektorja. Če imata vektorja *a* in *b* enako smer ali če je dolžina enega izmed vektorjev enaka nič, je njun vektorski produkt enak nič.

$$a \times b = \begin{bmatrix} y_1 * z_2 - z_1 * y_2 \\ z_1 * x_2 - x_1 * z_2 \\ x_1 * y_2 - y_1 * x_2 \end{bmatrix}, \text{kjer } a = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \text{ in } b = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} \tag{4.2}$$

Javanska metoda, ki implementira vektorski produkt, njeni testi in rezultati mutacijskega testiranja prvega in drugega reda mutantov so predstavljeni v *prilogi 2*. Metodo smo testirali tako, da smo vektorjema *a* in *b* določili koordinate z vrednostmi $x = 1$, $y = 1$, $z = 1$, pri čemer smo kot izhod pričakovali ničelni vektor ($x = 0$, $y = 0$, $z = 0$).

Mutacija metode je proizvedla skupno 36 mutantov prvega reda in 702 mutantov drugega reda. Mutacijska ocena testiranja vseh mutantov prvega reda je bila 75,0%, ocena testiranja mutantov drugega reda pa 85,0%.

4.3.3 Heronova formula

V ravninski geometriji je Heronova formula (4.3) namenjena računanju ploščine trikotnika s podanimi dolžinami stranic *a*, *b*, *c*. Spremenljivka *s* predstavlja polovični obseg ali semiperimeter (4.4) trikotnika.

$$P = \sqrt{s * (s-a) * (s-b) * (s-c)} \tag{4.3}$$

$$s = \frac{a+b+c}{2} \tag{4.4}$$

Javanska metoda, ki implementira Heronovo formulo, njeni testi in rezultati mutacijskega testiranja prvega in drugega reda mutantov so predstavljeni v *prilogi 3*. Metodo smo testirali tako, da smo na vhod podali vrednosti a = 2, b = 14 in c = 14 in kot izhod pričakovali izračunano vrednost ploščine P = 13,96.

Mutacija metode je proizvedla skupno 36 mutantov prvega reda in 691 mutantov drugega reda. Mutacijska ocena testiranja vseh mutantov prvega reda je bila 88,9%, ocena testiranja mutantov drugega reda pa 97,4%.

4.3.4 Presečišče premic

Koordinati točke, v kateri se sekata dve premici, ki ju v dvorazsežnem prostoru oziroma ravnini določata para točk, izračunamo po enačbi (4.5). Premico p_1 definirata točki $T_1(x_1, y_1)$ in $T_2(x_2, y_2)$, premico p_2 pa točki $T_3(x_3, y_3)$ in $T_4(x_4, y_4)$. Presečišče premice p_1 in p_2 je v točki (P_x, P_y).

$$(P_x, P_y) = \left(\frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_1 - x_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}, \frac{(x_1y_2 - y_1x_2)(y_3 - y_4) - (y_1 - y_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \right) \tag{4.5}$$

V primeru, da sta dve premici vzporedni je imenovalec v ulomku enak nič (4.6).

$$(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4) = 0 \text{, če } p_1 || p_2 \tag{4.6}$$

Javanska metoda, ki implementira formulo za presečišče dveh premic, njeni testi in rezultati mutacijskega testiranja prvega in drugega reda mutantov so predstavljeni v *prilogi 4*. Metodo smo testirali z dvema testoma. Prvi test metodi poda točki $T_1(-1, 1)$ in $T_2(1, -1)$, ki določata prvo premico, ter točki $T_3(-1, -1)$ in $T_4(1, 1)$, ki določata drugo premico. Kot rezultat pričakujemo, da je presečišče premic v točki $(0, 0)$. Drugi test preveri primer, ko sta

dve premici vzporedni. Metodi poda točki $T_1(1,2)$ in $T_2(5,2)$, ki določata prvo premico, ter točki $T_3(-3,7)$ in $T_4(2,7)$, ki določata drugo premico. Od metode pričakujemo, da v tem primeru ne vrne koordinat, saj presečišče ne obstaja.

Mutacija metode je proizvedla skupno 124 mutantov prvega reda in 9032 mutantov drugega reda. Mutacijska ocena testiranja vseh mutantov prvega reda je bila 54,0%, ocena testiranja mutantov drugega reda pa 74,9%.

4.3.5 Množenje matrik

Množenje matrik je binarna računska operacija, ki po posebnem postopku zmnoži dve matriki, pri čemer kot rezultat dobimo novo matriko. Množenje dveh matrik (4.7) je izvedljivo le, če je število stolpcev prve matrike enako številu vrstic druge matrike. Če je *A* matrika razsežnosti $m \times n$ (m vrstic, n stolpcev) in *B* matrika razsežnosti $n \times p$ (n vrstic, p stolpcev), potem je njun produkt *AB* matrika razsežnosti $m \times p$ (m vrstic, p stolpcev), katere elementi so skalarni produkt ustrezne vrstice matrike *A* in ustreznega stolpca matrike *B*.

$$[AB]_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \cdots + A_{i,n}B_{n,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j} \ , \quad \text{kjer je } 1 \leq \text{i} \leq \text{m in } 1 \leq \text{ j} \leq \text{p} \tag{4.7}$$

Javanska metoda, ki implementira postopek množenja dveh matrik, njeni testi in rezultati mutacijskega testiranja prvega in drugega reda mutantov so predstavljeni v *prilogi 5*. Metodo smo testirali tako, da smo kot vhod podali matriki $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ in $\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$, kot produkt pa na izhodu pričakovali matriko $\begin{bmatrix} 6 & 6 \\ 15 & 15 \end{bmatrix}$. Mutacija metode je proizvedla skupno 8 mutantov prvega reda in 19 mutantov drugega reda. Število generiranih mutantov je nizko, saj je bil postopek množenja matrik implementiran z zanko *for*, pri čemer sta bila za dejansko računanje uporabljena le dva aritmetična operatorja.

Mutacijska ocena testiranja vseh mutantov prvega reda je bila 87,5%, ocena testiranja mutantov drugega reda pa 100,0%.

4.3.6 Razdalja točke od premice

Razdalja točke T_0 od premice *p* (4.8) je dolžina daljice, ki poteka pravokotno od točke do premice. Pri tem je točka podana s koordinatami kot $T_0(x_0, y_0)$, premica pa predstavljena v implicitni obliki $ax + by + c = 0$.

$$d(T_0, p) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}} \tag{4.8}$$

Javanska metoda, ki implementira izračun razdalje točke od premice, njeni testi in rezultati mutacijskega testiranja prvega in drugega reda mutantov so predstavljeni v *prilogi 6*. Metodo smo testirali tako, da smo kot vhod podali točko $T_0(1, 1)$ in premico $x + y + 4 = 0$, kot izračunano razdaljo pa na izhodu pričakovali vrednost $d(T_0, p) = \frac{6}{\sqrt{2}}$.

Mutacija metode je proizvedla skupno 24 mutantov prvega reda in 289 mutantov drugega reda. Mutacijska ocena testiranja vseh mutantov prvega reda je bila 91,7%, ocena testiranja mutantov drugega reda pa 96,5%.

4.3.7 Ničle kvadratne funkcije

Koreni oziroma ničle kvadratne funkcije (4.10) so vrednosti spremenljivke *x*, kjer velja pogoj $f(x) = 0$. Kvadratna funkcija (4.9) ima lahko eno ali dve ničli, lahko pa je brez ničel.

$$f(x) = ax^2 + bx + c \tag{4.9}$$

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a} \tag{4.10}$$

Število *D*, ki nastopa pod kvadratnim korenom, imenujemo diskriminanta (4.11) in določa koliko ničel ima kvadratna funkcija oziroma kolikokrat graf kvadratne funkcije seka abscisno os (os *x*). Če je diskriminanta:

- pozitivna ($D > 0$), ima funkcija dve realni ničli oziroma graf funkcije seka os *x* v dveh točkah.
- enaka nič ($D = 0$), ima funkcija eno realno ničlo oziroma se graf funkcije dotika osi *x* v eni točki.
- negativna ($D < 0$), funkcija nima realnih ničel oziroma graf funkcije ne seka osi x. Ničli lahko izračunamo le v kompleksnem prostoru.

$$D = b^2 - 4ac \tag{4.11}$$

Javanska metoda, ki implementira postopek odkrivanja ničel kvadratne funkcije, njeni testi in rezultati mutacijskega testiranja prvega in drugega reda mutantov so predstavljeni v *prilogi 7*. Metodo smo testirali s tremi testi, pri čemer vsak test testira določeno stanje diskriminante. Prvi test obravnava primer, ko ima diskriminanta negativno vrednost oziroma funkcija nima ničel. Metodi kot vhod poda kvadratno funkcijo $f(x) = x^2 + 4x + 5$ in kot izhod ne pričakuje nobene vrednosti. Drugi test obravnava primer, ko je diskriminanta enaka

nič oziroma ima funkcija eno ničlo. Metodi kot vhod poda kvadratno funkcijo $f(x) = x^2$, kot rezultat pa pričakuje eno ničlo, in sicer $x_1 = 0$. Tretji test pa obravnava primer, ko je vrednost diskriminante pozitivna oziroma ima funkcija dve ničli. Metodi kot vhod poda kvadratno funkcijo $f(x) = x^2 - 2x - 3$, kot izhod pa pričakuje dve ničli, in sicer $x_1 = 3$ in $x_2 = -1$.

Mutacija metode je proizvedla skupno 44 mutantov prvega reda in 1059 mutantov drugega reda. Mutacijska ocena testiranja vseh mutantov prvega reda je bila 81,8%, ocena testiranja mutantov drugega reda pa 96,6%.

4.3.8 Strassenovo množenje matrik

Strassenov algoritem za množenje matrik temelji na strategiji deli in vladaj in v primerjavi z običajnim množenjem matrik nudi hitrejšo rešitev pri matrikah večjih dimenzij. To doseže z rekurzivnim razbitjem matrik velikosti $2^n \times 2^n$ na sedem podmatrik oziroma problem množenja matrik razdeli na sedem podproblemov istega tipa. Na ta način se zmanjša število operacij množenja in poveča število operacij seštevanja, ki so manj zahtevne.

Javanske metode, ki implementirajo Strassenov algoritem za množenje matrik, njihovi testi in rezultati mutacijskega testiranja prvega in drugega reda mutantov so predstavljeni v *prilogi 8*. Algoritem smo testirali tako, da smo glavni metodi *strassen* kot vhod podali dve enaki matriki, ki sta nosili vrednosti $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$, kot produkt matrik pa na izhodu pričakovali matriko z vrednostmi $\begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}$. Strassenov algoritem za razliko od ostalih programov obsega več metod, ki med seboj tesno sodelujejo. Postopek mutiranja smo zato izvedli znotraj posamezne metode in dobili mutante prvega reda. Mutante drugega reda pa smo dobili z vstavljanjem dveh napak znotraj posamezne metode (kjer je bilo to možno) in s kombiniranjem napak med metodami (dve metodi, vsaka vsebuje eno napako).

Mutacija metod je proizvedla skupno 68 mutantov prvega reda in 2152 mutantov drugega reda. Mutacijska ocena testiranja vseh mutantov prvega reda je bila 63,2%, ocena testiranja mutantov drugega reda pa 83,6%.

4.4 Rezultati mutacij prvega reda

Tabela 4.1 predstavlja skupen rezultat mutacijskega testiranja mutantov prvega reda, ki so nastali iz vseh osmih aritmetičnih programov omenjenih v poglavju 4.3. Celotni rezultati testiranja mutantov prvega reda za posamezen program so navedeni v prilogah.

Tabela je razdeljena na sedem stolpcev. Stolpec *Mutacije* ponazarja kategorije opisane v poglavju 4.2 oziroma operatorje, ki so nastali kot rezultat mutacije aritmetičnih operatorjev v originalnih programih, in kombinacije med njimi. Posamezen operator v stolpcu *Mutacije* predstavlja vse mutante, ki so nastali z mutacijo poljubnega aritmetičnega operatorja v definirani operator oziroma operatorje.

1. Stopnja	**Mutacije**	**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
	+	54	17	71	54/71	76,1%
	-	59	6	65	59/65	90,8%
	*	33	26	59	33/59	55,9%
	/	37	44	81	37/81	45,7%
	%	57	35	92	57/92	62,0%
	Skupaj	240	128	368	240/368	65,22%
	Povprečje					66,08%
2. Stopnja	**Mutacije**	**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
	+ -	113	23	136	113/136	83,1%
	+ *	87	43	130	87/130	66,9%
	+ /	91	61	152	91/152	59,9%
	+ %	111	52	163	111/163	68,1%
	- *	92	32	124	92/124	74,2%
	- /	96	50	146	96/146	65,8%
	- %	116	41	157	116/157	73,9%
	* /	70	70	140	70/140	50,0%
	* %	90	61	151	90/151	59,6%
	/ %	94	79	173	94/173	54,3%
	Skupaj	960	512	1472	960/1472	65,22%
	Povprečje					65,57%

3. Stopnja	**Mutacije**	**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
	+ - *	146	49	195	146/195	74,9%
	+ - /	150	67	217	150/217	69,1%
	+ - %	170	58	228	170/228	74,6%
	+ * /	124	87	211	124/211	58,8%
	+ * %	144	78	222	144/222	64,9%
	+ / %	148	96	244	148/244	60,7%
	- * /	129	76	205	129/205	62,9%
	- * %	149	67	216	149/216	69,0%
	- / %	153	85	238	153/238	64,3%
	* / %	127	105	232	127/232	54,7%
	Skupaj	1440	768	2208	1440/2208	65,22%
	Povprečje					65,38%

4. Stopnja	**Mutacije**	**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
	+ - * /	183	93	276	183/276	66,3%
	+ - * %	203	84	287	203/287	70,7%
	+ - / %	207	102	309	207/309	67,0%
	+ * / %	181	122	303	181/303	59,7%
	- * / %	186	111	297	186/297	62,6%
	Skupaj	960	512	1472	960/1472	65,22%
	Povprečje					65,28%

5. Stopnja	**Mutacije**	**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
	+ - * / %	240	128	368	240/368	65,22%
	Povprečje					65,22%

Tabela 4.1: Rezultati mutacij prvega reda.

Za boljše razumevanje sta navedena dva primera. (1) Mutacija "+" predstavlja vse mutante, ki so nastali pri mutaciji poljubnega aritmetičnega operatorja v operator seštevanja. (2) Mutacija "* / %" pa predstavlja vse mutante, ki so nastali pri mutaciji poljubnega aritmetičnega operatorja v operator množenja, operator deljenja in operator modula - torej je mutacija "* / %" kombinacija mutacij "*", "/" in "%", pri čemer je število mutantov vsota števila mutantov posamezne mutacije.

Stopnje določajo obseg kombinacij posameznih kategorij oziroma *mutacij*. Prva stopnja predstavlja rezultate vsake mutacije posebej, druga stopnja predstavlja rezultate vseh kombinacij parov mutacij, pri čemer je vsak par seštevek števila mutantov obeh mutacij, in peta stopnja predstavlja rezultat vseh mutacij skupaj oziroma rezultat celotne mutacijske analize. Namen stopenj je pregled učinkovitosti posameznega nabora mutacij v odvisnosti od števila mutantov, ki jih nabor vsebuje. Na ta način se lahko uporabnik odloči za določen nabor, ki po velikosti in učinkovitosti odkrivanja pomanjkljivosti v testih zadostuje njegovim potrebam.

Stolpec *Št. ubitih mutantov* predstavlja število mutantov, ki so jih testi zaznali in posledično ubili, stolpec *Št. živih mutantov* pa predstavlja število mutantov, katerih testi niso odkrili in so tako preživeli. Skupno število mutantov določene mutacije oziroma nabora mutacij predstavlja stolpec *Št. vseh mutantov*.

Zadnja dva stolpca prikazujeta enak podatek v različnih zapisih, in sicer mutacijsko oceno kot razmerje med številom ubitih mutantov in številom vseh mutantov in kot zapis z odstotki, ki bolj neposredno prikazuje učinkovitost oziroma težo določene mutacije.

Povprečje predstavlja povprečno mutacijsko oceno vseh naborov znotraj določene stopnje, *Skupaj* pa skupne vrednosti stolpcev znotraj določene stopnje. Zelena polja predstavljajo najmanjše število mutantov z najnižjo mutacijsko oceno, rdeča polja pa najmanjše število mutantov z najvišjo mutacijsko oceno.

Iz rezultatov je razvidno, da ima mutacija, ki spremeni poljubni aritmetični operator v operator deljenja (v nadaljevanju *mutacija v deljenje*) največjo težo, kar pomeni, da testi to mutacijo najtežje zaznajo. Posledično imajo tudi vsi nabori mutacij, ki vsebujejo *mutacijo v deljenje* večjo težo kot nabori, ki ne vsebujejo *mutacije v deljenje*. Nabori z največjo težo po stopnjah so:

1. mutacija v deljenje ("/"),
2. mutaciji v množenje in deljenje ("* /"),
3. mutacije v množenje, deljenje in modul ("* / %") in
4. mutacije v seštevanje, množenje, deljenje in modul ("+ * / %").

Za razliko od *mutacije v deljenje* pa ima mutacija, ki spremeni poljubni aritmetični operator v operator odštevanja (v nadaljevanju *mutacija v odštevanje*), najmanjšo težo, kar pomeni, da jo testi najlažje zaznajo in označijo delovanje mutanta za nepravilno. Vpliv najnižje teže *mutacije v odštevanje* se odraža tudi pri naborih, ki vsebujejo le-to. Nabori z najnižjo težo po stopnjah so:

1. mutacija v odštevanje ("-"),

2. mutaciji v seštevanje in odštevanje ("+ -"),

3. mutacije v seštevanje, odštevanje in množenje ("+ - *") in

4. mutacije v seštevanje, odštevanje, množenje in modul ("+ - * %").

Mutacijska ocena vrstice *Skupaj* se po stopnjah ne spreminja, saj znotraj posamezne stopnje nastopajo vse možne kombinacije mutacij, ki kljub ponavljanju ohranjajo enak delež ubitih in živih mutantov glede na skupno število mutantov. Prav tako je vrstica *Skupaj* prve stopnje enaka peti stopnji, saj je peta stopnja skupek oziroma seštevek vseh mutacij prve stopnje.

Vrednost *Povprečje* konvergira od prve stopnje proti povprečni vrednosti pete stopnje oziroma k celotni mutacijski oceni vseh mutacij. Do tega pojava pride, ker so nabori z večanjem stopnje vedno bolj zahtevni in podobni polnemu naboru mutacij ("+ - * / %").

Pridobljeni rezultati nam nudijo večji pregled učinkovitosti posameznih mutacij prvega reda in kombinacij med njimi. Na podlagi zahtevnosti programa, največjega števila ustvarjenih mutantov in najnižje sprejemljive skupne mutacijske ocene testov lahko izberemo ustrezen nabor mutacij, ki bo zadostoval našim zahtevam. Primera:

- Vkolikor želimo mutacijsko analizo opraviti s čim manjšim številom mutantov in hkrati postaviti najstrožji kriterij, bomo uporabili mutacijo "/".

- Večjo natančnost lahko dosežemo z večanjem nabora in posledično tudi števila mutantov. Tako lahko z naborom mutacij "* / %" postavimo strog kriterij, ki je po mutacijski oceni dober približek polnemu naboru mutacij, hkrati pa ustvarimo tretjino manj mutantov. S tem prihranimo 33% časa pri testiranju mutantov.

Bolj zahtevni in obsežni programi zahtevajo večji nabor mutacij, saj se njihovi operatorji pojavljajo v večjem obsegu in v različnem kontekstu. Zato je izbor ustreznega nabora mutacij toliko bolj pomemben.

Pri izvajanju mutacijske analize je priporočljiva izbira naborov, ki vsebujejo mutacijo "/", saj ne glede na njihovo velikost zagotavljajo največji delež preživelih mutantov in tako najučinkoviteje odkrijejo pomanjkljivosti v testnih primerih.

Nabori, ki vsebujejo mutacijo "+", za uporabo pri testiranju niso primerni, saj ustvarjene mutante največkrat odkrijejo že najosnovnejši testni primeri, ki so očitno pomanjkljivi. Naš cilj je namreč odkriti čim več pomanjkljivosti v testih in ne s slabimi testi ubiti velikega števila mutantov.

4.5 Rezultati mutacij drugega reda

Tabela 4.2 predstavlja skupen rezultat mutacijskega testiranja mutantov drugega reda, ki so nastali iz vseh osmih aritmetičnih programov omenjenih v poglavju 4.3. Vsak mutant predstavlja spremembi dveh aritmetičnih operatorjev originalnega programa. Celotni rezultati testiranja mutantov drugega reda za posamezen program so navedeni v prilogah.

Tabela je razdeljena na sedem stolpcev. Stolpec *Mutacije prvega reda* ponazarja operatorje, ki so nastali kot rezultat prve mutacije oziroma prve spremembe aritmetičnih operatorjev v originalnih programih. Stolpec *Mutacije drugega reda* pa ponazarja operatorje, ki so nastali z vpeljavo druge spremembe na originalne programe oziroma z vpeljavo mutacije na mutante prvega reda mutacij.

Mutacije prvega reda	**Mutacije drugega reda**	**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
+	+	1030	146	1176	1030/1176	87,6%
	-	670	69	739	670/739	90,1%
	*	573	152	725	573/725	79,0%
	/	945	210	1155	945/1155	81,8%
	%	1013	252	1265	1013/1265	80,1%
	Skupaj	4231	829	5060	4231/5060	83,62%
	Povprečje					83,84%
-	+	670	69	739	670/739	90,1%
	-	593	21	614	593/614	96,6%
	*	551	62	613	551/613	89,9%
	/	769	66	835	769/835	92,1%
	%	834	88	922	834/922	90,5%
	Skupaj	3417	306	3723	3417/3723	91,78%
	Povprečje					91,94%
*	+	573	152	725	573/725	79,0%
	-	551	62	613	551/613	89,9%
	*	374	204	578	374/578	64,7%
	/	545	270	815	545/815	66,9%
	%	614	283	897	614/897	68,5%
	Skupaj	2657	971	3628	2657/3628	73,24%
	Povprečje					73,79%

Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
/	+	945	210	1155	945/1155	81,8%
	-	769	66	835	769/835	92,1%
	*	545	270	815	545/815	66,9%
	/	813	396	1209	813/1209	67,2%
	%	934	398	1332	934/1332	70,1%
	Skupaj	4006	1340	5346	4006/5346	74,93%
	Povprečje					75,63%
%	+	1013	252	1265	1013/1265	80,1%
	-	834	88	922	834/922	90,5%
	*	614	283	897	614/897	68,5%
	/	934	398	1332	934/1332	70,1%
	%	1074	398	1472	1074/1472	73,0%
	Skupaj	4469	1419	5888	4469/5888	75,90%
	Povprečje					76,41%

Tabela 4.2: Rezultati mutacij drugega reda - vse kombinacije mutacij drugega reda.

	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
Skupaj	11332	3015	14347	11332/14347	78,99%
Povprečje					78,99%

Tabela 4.3: Rezultati mutacij drugega reda - skupna vrednost vseh kombinacij mutacij drugega reda brez ponavljanja.

Par operatorjev v stolpcu *Mutacije prvega reda* in *Mutacije drugega reda* predstavlja vse mutante, ki so nastali z mutacijo poljubnega prvega aritmetičnega operatorja v definirani operator prvega reda in z mutacijo poljubnega drugega aritmetičnega operatorja v definirani operator drugega reda. Pri tem je treba upoštevati, da druga sprememba ne sme povoziti prve spremembe, kot je opisano v poglavju 4.2. Torej mutacija "+, /" predstavlja vse mutante, ki so nastali z vpeljavo dveh sprememb oziroma mutacij, ki poljuben aritmetičen operator spremenijo v operator seštevanja in drug poljuben aritmetičen operator spremenijo v operator deljenja.

Stolpec *Št. ubitih mutantov* predstavlja število mutantov, ki so jih testi zaznali in posledično ubili, stolpec *Št. živih mutantov* pa predstavlja število mutantov, katerih testi niso

odkrili in so tako preživeli. Število mutantov, ki ga je ustvaril določen par mutacij oziroma določena mutacija drugega reda, predstavlja stolpec *Št. vseh mutantov*.

Zadnja dva stolpca prikazujeta enak podatek v različnih zapisih, in sicer mutacijsko oceno kot razmerje med številom ubitih mutantov in številom vseh mutantov in kot zapis z odstotki, ki bolj neposredno prikazuje učinkovitost oziroma težo določene mutacije drugega reda.

Povprečje predstavlja povprečno mutacijsko oceno vseh mutacij drugega reda, ki so izvirale iz določene mutacije prvega reda, *Skupaj* pa skupne vrednosti stolpcev mutacij drugega reda. Zelena polja predstavljajo najmanjše število mutantov z najnižjo mutacijsko oceno, rdeča polja pa najmanjše število mutantov z najvišjo mutacijsko oceno.

Rezultati kažejo, da imajo največjo težo mutacije drugega reda, katerih izvor je mutacija prvega reda, ki poljubni operator v originalnih programih spremeni v operator množenja, kar pomeni, da testi le-te najtežje odkrijejo. Prav tako ima mutacija "*, *" izmed vseh mutacij drugega reda najnižjo mutacijsko oceno, torej z njo postavimo najtežji kriterij pri vrednotenju testov.

Nasprotje pa predstavljajo vse mutacije drugega reda, katerih izvor je mutacija prvega reda, ki poljubni operator v originalnih programih spremeni v operator odštevanja. Le-te imajo najmanjšo težo, kar pomeni, da jih testi najlažje zaznajo. Mutacija "-, -" ima tako izmed vseh mutacij drugega reda najvišjo mutacijsko oceno, torej z njo postavimo najbolj blag kriterij pri vrednotenju testov.

Rezultati so sprva pokazali določeno odstopanje oziroma nepravilnosti pri številu ubitih in preživelih mutantov nekaterih mutacij drugega reda. Predpostavljali smo, da naj bi bila mutacija "*x*, *y*" (*x* je poljubna mutacija prvega reda in *y* poljubna mutacija drugega reda) po številu ubitih in preživelih mutantov enaka mutaciji "*y*, *x*", saj kljub različnemu zaporedju vstavljanja napak generirata enako število mutantov, ki so po vsebini prav tako enaki. Vendar temu ni tako.

Problem je predstavljal sistem MuJava, ki pri različnih mutacijah na različen način daje prednost računskim operacijam oziroma odvzema ali postavlja oklepaje, ki določajo prednost aritmetičnih operacij. Posledično to dejanje vpliva na število ubitih in preživelih mutantov in na rezultat mutacijske analize. Odstopanja smo zato ignorirali in sledili naši predpostavki, da mutacija drugega reda "*x*, *y*" producira enake mutante kot njena zrcalna mutacija "*y*, *x*", iz česar sledi "*x*, *y*" ≡ "*y*, *x*".

Tabela 4.3 prikazuje skupne rezultate vseh mutacij drugega reda, pri čemer upoštevamo, da je mutacija "*x*, *y*" enaka mutaciji "*y*, *x*", torej zanemarimo ponavljanje mutantov. Rezultat tako obsega mutacije:

- "+, +", "+, -", "+, *", "+, /", "+, %"

- "-, -", "-, *", "-, /", "-, %"
- "*, *", "*, /", "*, %"
- "/, /", "/, %" in
- "%, %".

Pridobljeni rezultati nam nudijo natančni pregled učinkovitosti posameznih mutacij drugega reda. Na podlagi zahtevnosti programa, največjega števila ustvarjenih mutantov in najnižje sprejemljive skupne mutacijske ocene testov lahko izberemo ustrezen par mutacij, ki bo zadostoval našim zahtevam. Primera:

- Vkolikor želimo mutacijsko analizo opraviti s čim manjšim številom mutantov in hkrati postaviti najstrožji kriterij, uporabimo mutacijo "*, *".
- Za natančnejši vzorec lahko uporabimo mutacijo "/, /", ki ponuja zelo podoben rezultat kot mutacija "*, *", a uporabi dvakrat več mutantov. Čas testiranja je tako daljši, vendar lahko pri zahtevnejših programih zagotovi bolj natančen rezultat.

Število mutantov, ki jih ustvarijo mutacije drugega reda, lahko zelo hitro uide izpod nadzora in posledično tudi čas testiranja le-teh. Temu se lahko izognemo s skrbno izbiro ustreznih mutacij. Pri izbiri mutacij je pomembno razmerje med težo kriterija (mutacijska ocena) in številom vseh mutantov.

Za testiranje je najbolj priporočljiv sklop mutacij "*, *x*" oziroma "*x*, *" (*x* je poljubna mutacija), saj le-ta ponuja najstrožji kriterij glede na relativno majhno število vseh mutantov, ki jih mutacije ustvarijo. Zagotavlja hitro in učinkovito vrednotenje testov.

Sklop mutacij "-, *x*" oziroma "*x*, -" prav tako ustvari nizko število mutantov, vendar ni učinkovit pri odkrivanju napak v testnih primerih in zato ni primerna izbira pri testiranju mutantov.

Za potrebe testiranja lahko izberemo poljubne mutacije drugega reda in si sami ustvarimo sklop mutacij, ki bo najbolje ovrednotil naše teste. Pri tem je priporočljivo vključevanje mutacij z nižjo mutacijsko oceno oziroma s strožjim kriterijem.

4.6 Ugotovitve

Več mutantov načeloma pomeni bolj natančen odraz stopnje učinkovitosti testov. Naša predpostavka pa trdi, da je smiselno število mutantov omejeno in je zato nepremišljen izbor vseh mutantov nesmiseln, saj obstaja manjše število mutantov, ki zagotovijo enak ali zelo podoben rezultat.

Selektivni izbor ustreznih mutacijskih operatorjev je eden izmed načinov, kako znižati število generiranih mutantov in hkrati ohraniti zadovoljivo stopnjo učinkovitosti mutacijske analize.

Rezultati naše analize nam podajajo nov način, kako pristopiti k mutacijski analizi. Dodatno optimizacijo lahko dosežemo še s selektivnim izborom mutacij, ki jih posamezen mutacijski operator izvede nad programsko kodo. Tako smo z razčlenitvijo mutacijskega operatorja AOR dokazali, da so nekatere mutacije bolj učinkovite od drugih in zato lahko nabor mutacij, ki jih uporablja operator AOR zmanjšamo glede na potrebe uporabnika. S tem znižamo časovno zahtevnost mutacijske analize, ki pri obsežnejših programih hitro uide izpod nadzora.

Uporabnik lahko na podlagi naše statistike izbere primeren nabor mutacij, ki po številu ustvarjenih mutantov in po ustreznosti mutacijske ocene zadostuje željenemu kriteriju odkrivanju napak v programski kodi. Mutacije z večjo težo oziroma nižjo mutacijsko oceno tako postavljajo strožji kriterij, medtem ko mutacije z nižjo težo oziroma višjo mutacijsko oceno postavljajo bolj blag kriterij.

Pri mutacijah prvega reda smo ugotovili, da:

1. najtežji kriterij prvega reda mutacij postavljajo mutanti, ki so nastali z mutacijo, ki poljuben aritmetičen operator programa spremenijo v operator deljenja in je zato le-ta najbolj učinkovit del mutacijskega operatorja AOR. Priporočljiva je za uporabo tako samostojno kot del nabora, ki vključuje tudi eno ali več ostalih mutacij.

2. mutante, ki so nastali z mutacijo, ki poljuben aritmetičen operator programa spremenijo v operator odštevanja, testi najlaže zaznajo in ubijejo, zato samostojna uporaba te mutacije ni priporočljiva razen v naboru z močnejšimi mutacijami.

Pri mutacijah drugega reda smo ugotovili, da:

1. je najbolj učinkovita in hkrati postavlja tudi najstrožji kriterij mutacija, ki v programu zamenja dva poljubna aritmetična operatorja z operatorjem množenja. Tako je tudi celoten sklop mutacij drugega reda, ki kot prvo napako v programu zamenjajo poljuben aritmetičen operator z operatorjem množenja, nabor z največjo težo.

2. podobno kot pri mutacijah prvega reda tudi pri mutacijah drugega reda predstavljajo najšibkejši kriterij mutacije, ki kot prvo vstavljeno napako zamenjajo poljuben aritmetičen operator z operatorjem odštevanja. Najmanj učinkovita mutacija je tako mutacija, ki v programu zamenja dva poljubna aritmetična operatorja z operatorjem odštevanja in ni priporočljiva za samostojno uporabo.

Pregled nad učinkovitostjo posameznih mutacij, ki jih vključuje mutacijski operator AOR, nam lahko pospeši postopek vrednotenja testov programov aritmetične narave. Skupna mutacijska ocena posamezne mutacije ali nabora mutacij nam pove, kako močan kriterij postavimo pri odkrivanju nedoslednosti v naših testih. Pri nadaljnjem testiranju lahko s pomočjo podatkov, ki so predstavljeni v tabeli 4.1 in 4.2, natančneje izberemo želene mutacije in mutacijsko analizo prilagodimo našim potrebam.

Pred testiranjem v stolpcu *Mutacijska ocena* določimo mutacijske ocene, ki najbolje predstavljajo zahtevnost želenega kriterija. Pri tem upoštevamo, da polja, pobarvana z zeleno barvo, predstavljajo najmanjše število mutantov z najnižjo mutacijsko oceno, polja, pobarvana z rdečo barvo, pa predstavljajo najmanjše število mutantov z najvišjo mutacijsko oceno.

Po določitvi ustreznih mutacijskih ocen, za izbrane mutacijske ocene pogledamo stolpec *Št. vseh mutantov* in izločimo mutacije, ki presegajo največje dovoljeno število ustvarjenih mutantov. Pri tem moramo upoštevati, da so števila v stolpcu *Št. vseh mutantov* relativna glede na skupno število mutantov, ki jih program ob uporabi mutacijskega operatorja AOR ustvari.

Dobljene vrstice predstavljajo mutacije oziroma nabore mutacij, ki najbolje ustrezajo našim testnim zahtevam. Izbrane mutacije nato kot del mutacijskega operatorja AOR uporabimo za ustvarjanje mutantov in vrednotenje testov. Če je bila naša izbira primerna, smo za postopek mutacijske analize porabili manj mutantov in časa ter dobili zelo dober približek rezultata, ki bi ga sicer dobili z uporabo vseh mutacij mutacijskega operatorja AOR.

Poglavje 5 Sklepne ugotovitve

V diplomskem delu smo se poglobili v koncept in delovanje mutacijskega testiranja programske opreme, ki je bilo idealno izhodišče za ustvarjanje prispevka k potencialni optimizaciji enega izmed najbolj učinkovitih testnih pristopov, zlasti na področju selektivnosti pri generiranju mutantov.

Po seznanitvi z osnovnimi pojmi in funkcionalnostmi mutacijskega testiranja smo preučili sistem MuJava, s katerim smo postavili okoliščine našega eksperimenta in zbirali podatke za statistično analizo učinkovitosti mutacijskega operatorja AOR.

Ustvarili smo osem različnih aritmetičnih programov, katere smo nato mutirali z mutacijskim operatorjem AOR, ki nadomesti aritmetične operatorje, kot rezultat pa dobili množico mutantov prvega reda in še večjo množico mutantov drugega reda. Mutante smo nato razdelili glede na učinek, ki ga je imela mutacija. Le-te smo v različnih kombinacijah testirali s testi, ki so bili šibke zasnove, kar pomeni, da so pravilno delovanje programa dokazali na podlagi le enega testnega primera in zagotovili stoodstotno pokritost programske kode.

Rezultate testiranja prvega in drugega reda mutacij smo beležili za vsak posamezni program posebej, na koncu pa smo rezultate združili v eno samo preglednico in jih analizirali. Končna slika analize je prikazala učinkovitost posamezne mutacije in kombinacij med mutacijami, ki jih izvede mutacijski operator AOR. S pomočjo analize tako lahko izberemo ustrezne mutacije glede na naš kriterij in zahteve ter na ta način v bodoče zmanjšamo število generiranih mutantov.

Selektivna mutacija temelji na principu izbora čim manj in čimbolj učinkovitih mutacijskih operatorjev, s čimer zmanjšamo število ustvarjenih mutantov. Na podlagi izvedene analize in pridobljenih rezultatov bi lahko pristop selektivne mutacije razširili, in sicer z uvedbo dvo-nivojske selekcije. Prvi nivo selekcije bi določal učinkovitost mutacijskih operatorjev, drugi nivo pa učinkovitost posameznih vrst mutacij, ki jih mutacijski operator premore. Na ta način bi bile mutacije bolj prilagojene posameznim tipom programov, poleg tega pa bi se število mutantov dodatno znižalo, hkrati pa bi ponujali bolj ključne in natančne rezultate.

V nadaljnjih raziskavah bi želeli podobno analizo opraviti tudi na drugih mutacijskih operatorjih in sčasoma uvesti splošna pravila za primernost uporabe posameznih operatorjev in mutacij nad ustaljenimi tipi programov.

Literatura

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton in F. G. Sayward, "Mutation Analysis", Georgia Institute of Technology Atlanta School of Information and Computer Science, Atlanta, GA, ZDA, 1979.

[2] K. Adamopoulos, M. Harman in R. M. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution", v *AAAI Genetic and Evolutionary Computation Conference 2004 (GECCO 2004)*, zv. 3103, str. 1338-1349, Seattle, WA, ZDA, 26.-30. junij 2004.

[3] R. T. Alexander, J. M. Bieman, J. Bixia in S. Ghosh, "Mutation of Java objects", v zborniku *13th International Symposium on Software Reliability Engineering*, *2002. ISSRE 2003.*, Fort Collins, CO, ZDA, 2002.

[4] P. Ammann in J. A. Offutt, "Syntax-Based Testing", v knjigi *Introduction to Software Testing*, Cambridge University Press, str. 170-212, New York, NY, ZDA, 2008.

[5] J. H. Andrews, L. C. Briand in Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?", v zborniku *Proceedings of the 27th International Conference on Software Engineering*, IEEE, str. 402-411, St. Louis, MO, ZDA, 2005.

[6] D. Baldwin in F. Sayward, "Heuristics for Determining Equivalence of Program Mutations," Yale University, Dept. of Computer Science, New Haven, CT, ZDA, 1979.

[7] Bernoullijeva enačba. (21, oktober 2013). *Wikipedija* [Online]. Dosegljivo: https://sl.wikipedia.org/wiki/Bernoullijeva_ena%C4%8Dba. [Dostopano 28. december 2014].

[8] H. Coles. PIT Mutation Testing. (26. november 2014). *PIT Mutation Testing* [Online]. Dosegljivo: http://pitest.org. [Dostopano 2. december 2014].

[9] H. Coles. PIT Mutation Testing: Basic Concepts. (26. november 2014). *PIT Mutation Testin*g [Online]. Dosegljivo: http://pitest.org/quickstart/basic_concepts/. [Dostopano 2. december 2014].

[10] M. W. Craf in J. A. Offutt, "Using Compiler Optimization Techniques to Detect Equivalent Mutants", v *The Journal of Software Testing, Verification and Reliability*, št. 3, zv. 4, str. 131-154, 27. december 1994.

[11] S. Danicic, M. Harman in R. Hierons, "Using Program Slicing to Assist in the Detection of Equivalent Mutants", v *Software Testing, Verification and Reliability*, št. 4, zv. 9, str. 233-262, december 1999.

[12] R. A. DeMillo, R. J. Lipton in F. G. Saywar, "Hints on Test Data Selection: Help for the Practicing Programmer", v *Computer*, št. 4, zv. 11, str. 34-41, april 1978.

[13] I. García-Rodrígue, M. Piattini in M. Polo, "Decreasing the cost of mutation", v *Software Testing, Verification and Reliability*, št. 2, zv. 19, str. 111-131, 2009.

[14] B. J. M. Grün, D. Schuler in A. Zeller, "The Impact of Equivalent Mutants", v zborniku *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, IEEE Computer Society, Washington, DC, ZDA, 2009.

[15] M. Hafiz. Mutation Testing Tool For Java. (25. september 2008). *Munawar Hafiz* [Online]. Dosegljivo: http://www.munawarhafiz.com/research/mutationtesting/index.htm. [Dostopano 25. november 2014].

[16] M. Harman, Y. Jia in X. Yao, "A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence", v zborniku *Proceedings of the 36th International Conference on Software Engineering*, ACM, str. 919-930, Hyderabad, IN, USA, 2014.

[17] M. Harman in Y. Jia, "An analysis and survey of the development of mutation testing", v *IEEE Transactions on Software Engineering*, IEEE, št. 5, zv. 37, str. 649-678, 29. september 2011.

[18] M. J. Harrold, J. A. Offutt in R. H. Untch, "Mutation analysis using mutant schemata", v *ACM SIGSOFT Software Engineering Notes*, št. 3, zv. 18, str. 139-148, New York, NY, ZDA, julij 1993.

[19] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets", v *IEEE Transactions on Software Engineering*, št. 4, zv. 8, str. 371-379, julij 1982.

[20] jUnit. (14. marec 2013). *SourceForge, Dice Holdings, Inc.* [Online]. Dosegljivo: http://sourceforge.net/projects/junit/. [Dostopano 26. november 2014].

[21] Y. R. Kwon, Y. S. Ma in J. A. Offutt, "MuJava: An Automated Class Mutation System", v *Software Testing, Verification and Reliability*, str. 97-133, junij 2005.

[22] Y. R. Kwon, Y. S. Ma in J. A. Offutt, "MuJava: A Mutation System for Java", v *Workshop on Automation of Software Test (AST 2006),* str. 78-84, maj 2006.

[23] N. Li in J. A. Offutt. µJava Home Page. (junij 2013). *µJava* [Online]. Dosegljivo: http://www.cs.gmu.edu/~offutt/mujava/. [Dostopano 24. november 2014].

[24] Y. S. Ma in J. A. Offutt. Description of Class Mutation Mutation Operators for Java. (1. avgust 2014). *µJava* [Online]. Dosegljivo: http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf. [Dostopano 12. december 2014].

[25] L. Madeyski in N. Radyk, "Judy - a mutation testing tool for java", v *Software, IET*, št. 1, zv. 4, str. 32-42, februar 2010.

[26] I. Moore. Jester – the JUnit test tester. (7. november 2005). *SourceForge, Dice Holdings, Inc.* [Online]. Dosegljivo: http://jester.sourceforge.net/. [Dostopano 2. december 2014].

[27] R. Niedermayr, "Meaningful and Practical Measures for Regression Test Reliability", Fakultät für Informatik, Der Technischen Universität München, München, Nemčija, 13. september 2013.

[28] J. A. Offutt in J. Pan, "Detecting equivalent mutants and the feasible path problem", v zborniku *Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on 17-21 Jun 1996*, str. 224 - 236, Gaithersburg, MD, ZDA, 1996.

[29] J. A. Offutt in R. H. Untch, "Mutation 2000: Uniting the orthogonal", v *Mutation testing for the new century*, Kluwer Academic Publishers, str. 34-44, Norwell, MA, ZDA, 2001.

[30] Reel Two. Jumble. (6. junij 2007). *SourceForge, Dice Holdings, Inc.* [Online]. Dosegljivo: http://jumble.sourceforge.net/mutations.html. [Dostopano 2. december 2014].

[31] I. Rožanc, "Uvod v testiranje", v *Študijsko gradivo za interno uporabo pri predmetu Testiranje in kakovost (TiK)*, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, Ljubljana, Slovenija, 2013.

[32] B. Smith Hatfield in L. Williams. MuClipse - An open source mutation testing plug-in for Eclipse. (29. september 2011). *SourceForge, Dice Holdings, Inc.* [Online]. Dosegljivo: http://muclipse.sourceforge.net/. [Dostopano 2. december 2014].

[33] B. Smith Hatfield. MuClipse: Mutation Testing for Eclipse. (13. januar 2007). *NC State University* [Online]. Dosegljivo: https://www.csc.ncsu.edu/academics/undergrad/honors/thesis/muclipsebinder.pdf. [Dostopano 2. december 2014].

[34] B. Smith Hatfield in L. Williams, "On Guiding the Augmentation of an Automated Test Suite via Mutation Analysis", v *Empirical Software Engineering*, št. 3, zv. 14, str. 341-369, 2009.

[35] M. Umar, "An Evaluation of Mutant Operators For Equivalent Mutants", Department of Computer Science, King's College, London, Anglija, 2006.

Priloge

Priloga 1 Bernoullijeva enačba

I. Program

```
/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class BernoullisEquationDensity {

    /**
     * g = gravitational acceleration
     * p = pressure
     * v = velocity
     * h = height
     * D = density
     * @return D
     */
    public double bernoullisEquationDensity(double p1, double v1, double h1,
                                double p2, double v2, double h2) {
        double g = 9.80665;
        double pressureDiff = p2 - p1;
        double velocityDiff = (Math.pow(v1, 2) - Math.pow(v2, 2))/2;
        double heightDiff = h1 - h2;
        double density = pressureDiff/(velocityDiff + g*heightDiff);

        return density;
    }
}
```

II. Test

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class BernoullisEquationDensityTest {

    private BernoullisEquationDensity bed;
    private static final double DELTA = 1e-15;

    public BernoullisEquationDensityTest() {
    }

    @Before
    public void setUp() {
        bed = new BernoullisEquationDensity();
    }

    @After
    public void tearDown() {
        bed = null;
    }

    @Test
    public void testBernoullisEquationDensity() {
        double result = bed.bernoullisEquationDensity(0.0, 1.0, 1.0, 0.0, 0.0,
                                                      0.0);
        assertEquals(0.0, result, DELTA);
    }
}
```

III. Rezultati testiranja mutantov prvega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

1. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	1	5	6	1/6	16.6%
	-	1	3	4	1/4	25.0%
	*	0	6	6	0/6	0.0%
	/	1	4	5	1/5	20.0%
	%	3	4	7	3/7	42.9%
	Skupaj	6	22	28	6/28	21.43%
	Povprečje					20.91%

2. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ -	2	8	10	2/10	20.0%
	+ *	1	11	12	1/12	8.3%
	+ /	2	9	11	2/11	18.2%
	+ %	4	9	13	4/13	30.8%
	- *	1	9	10	1/10	10.0%
	- /	2	7	9	2/9	22.2%
	- %	4	7	11	4/11	36.4%
	* /	1	10	11	1/11	9.1%
	* %	3	10	13	3/13	23.1%
	/ %	4	8	12	4/12	33.3%
	Skupaj	24	88	112	24/112	21.43%
	Povprečje					21.14%

3. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - *	2	14	16	2/16	12.5%
	+ - /	3	12	15	3/15	20.0%
	+ - %	5	12	17	5/17	29.4%
	+ * /	2	15	17	2/17	11.8%
	+ * %	4	15	19	4/19	21.1%
	+ / %	5	13	18	5/18	27.8%
	- * /	2	13	15	2/15	13.3%
	- * %	4	13	17	4/17	23.5%
	- / %	5	11	16	5/16	31.3%
	* / %	4	14	18	4/18	22.2%
	Skupaj	36	132	168	36/168	21.43%
	Povprečje					21.28%

4. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * /	3	18	21	3/21	14.3%
	+ - * %	5	18	23	5/23	21.7%
	+ - / %	6	16	22	6/22	27.3%
	+ * / %	5	19	24	5/24	20.8%
	- * / %	5	17	22	5/22	22.7%
	Skupaj	24	88	112	24/112	21.43%
	Povprečje					21.37%

5. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * / %	6	22	28	6/28	21.43%
	Povprečje					21.43%

IV. Rezultati testiranja mutantov drugega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno
Najmanjše število mutantov z najvišjo mutacijsko oceno

Vse kombinacije mutacij drugega reda	Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	+	10	20	30	10/30	33.3%
		-	8	13	21	8/21	38.1%
		*	5	26	31	5/31	16.1%
		/	9	17	26	9/26	34.6%
		%	18	18	36	18/36	50.0%
		Skupaj	50	94	144	50/144	34.72%
		Povprečje					34.43%
	-	+	8	13	21	8/21	38.1%
		-	6	6	12	6/12	50.0%
		*	5	16	21	5/21	23.8%
		/	8	10	18	8/18	44.4%
		%	15	9	24	15/24	62.5%
		Skupaj	42	54	96	42/96	43.75%
		Povprečje					43.77%
	*	+	5	26	31	5/31	16.1%
		-	5	16	21	5/21	23.8%
		*	6	24	30	6/30	20.0%
		/	8	18	26	8/26	30.8%
		%	18	18	36	18/36	50.0%
		Skupaj	42	102	144	42/144	29.17%
		Povprečje					28.14%

	/	+	9	17	26	9/26	34.6%
		-	8	10	18	8/18	44.4%
		*	8	18	26	8/26	30.8%
		/	10	10	20	10/20	50.0%
		%	18	12	30	18/30	60.0%
		Skupaj	53	67	120	53/120	44.17%
		Povprečje					43.97%
	%	+	18	18	36	18/36	50.0%
		-	15	9	24	15/24	62.5%
		*	18	18	36	18/36	50.0%
		/	18	12	30	18/30	60.0%
		%	30	12	42	30/42	71.4%
		Skupaj	99	69	168	99/168	58.93%
		Povprečje					58.79%
Skupna vrednost vseh kombinacij mutacij brez ponavljanja			**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
		Skupaj	174	229	403	174/403	43.18%
		Povprečje					43.18%

Priloga 2 Vektorski produkt

I. Program

```
/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class CrossProduct {

    public double[] crossProduct(double x1, double y1, double z1, double x2,
                          double y2, double z2) {

        double x = y1*z2 - z1*y2;
        double y = z1*x2 - x1*z2;
        double z = x1*y2 - y1*x2;

        return new double[]{x, y, z};
    }
}
```

II. Test

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class CrossProductTest {

    private CrossProduct cp;
    private static final double DELTA = 1e-15;

    public CrossProductTest() {
    }

    @Before
    public void setUp() {
        cp = new CrossProduct();
    }

    @After
    public void tearDown() {
        cp = null;
    }

    @Test
    public void testCrossProduct() {
        double[] result = cp.crossProduct(1.0, 1.0, 1.0, 1.0, 1.0, 1.0);
        double[] expect = {0.0, 0.0, 0.0};
        assertEquals(expect[0], result[0], DELTA);
        assertEquals(expect[1], result[1], DELTA);
        assertEquals(expect[2], result[2], DELTA);
    }
}
```

III. Rezultati testiranja mutantov prvega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

1. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	9	0	9	9/9	100.0%
	-	6	0	6	6/6	100.0%
	*	3	0	3	3/3	100.0%
	/	3	6	9	3/9	33.3%
	%	6	3	9	6/9	66.6%
	Skupaj	27	9	36	27/36	75.00%
	Povprečje					80.00%

2. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ -	15	0	15	15/15	100.0%
	+ *	12	0	12	12/12	100.0%
	+ /	12	6	18	12/18	66.6%
	+ %	15	3	18	15/18	83.3%
	- *	9	0	9	9/9	100.0%
	- /	9	6	15	9/15	60.0%
	- %	12	3	15	12/15	80.0%
	* /	6	6	12	6/12	50.0%
	* %	9	3	12	9/12	75.0%
	/ %	9	9	18	9/18	50.0%
	Skupaj	108	36	144	108/144	75.00%
	Povprečje					76.50%

3. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - *	18	0	18	18/18	100.0%
	+ - /	18	6	24	18/24	75.0%
	+ - %	21	3	24	21/24	87.5%
	+ * /	15	6	21	15/21	71.4%
	+ * %	18	3	21	18/21	85.7%
	+ / %	18	9	27	18/27	66.6%
	- * /	12	6	18	12/18	66.6%
	- * %	15	3	18	15/18	83.3%
	- / %	15	9	24	15/24	62.5%
	* / %	12	9	21	12/21	57.1%
	Skupaj	162	54	216	162/216	75.00%
	Povprečje					75.59%

4. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * /	21	6	27	21/27	77.7%
	+ - * %	24	3	27	24/27	88.8%
	+ - / %	24	9	33	24/33	72.7%
	+ * / %	21	9	30	21/30	70.0%
	- * / %	18	9	27	18/27	66.6%
	Skupaj	108	36	144	108/144	75.00%
	Povprečje					75.21%

5. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * / %	27	9	36	27/36	75.00%
	Povprečje					75.00%

IV. Rezultati testiranja mutantov drugega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

Vse kombinacije mutacij drugega reda	Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	+	72	0	72	72/72	100.0%
		-	42	6	48	42/48	87.5%
		*	24	0	24	24/24	100.0%
		/	72	0	72	72/72	100.0%
		%	66	6	72	66/72	91.7%
		Skupaj	276	12	288	276/288	95.83%
		Povprečje					95.83%
	-	+	42	6	48	42/48	87.5%
		-	30	0	30	30/30	100.0%
		*	12	6	18	12/18	66.7%
		/	42	6	48	42/48	87.5%
		%	42	6	48	42/48	87.5%
		Skupaj	168	24	192	168/192	87.50%
		Povprečje					85.83%
	*	+	24	0	24	24/24	100.0%
		-	12	6	18	12/18	66.7%
		*	6	0	6	6/6	100.0%
		/	24	0	24	24/24	100.0%
		%	18	6	24	18/24	75.0%
		Skupaj	84	12	96	84/96	87.50%
		Povprečje					88.33%

	/	+	72	0	72	72/72	100.0%
		-	42	6	48	42/48	87.5%
		*	24	0	24	24/24	100.0%
		/	42	30	72	42/72	58.3%
		%	51	21	72	51/72	70.8%
		Skupaj	231	57	288	231/288	80.21%
		Povprečje					83.33%
	%	+	66	6	72	66/72	91.7%
		-	42	6	48	42/48	87.5%
		*	18	6	24	18/24	75.0%
		/	51	21	72	51/72	70.8%
		%	54	18	72	54/72	75.0%
		Skupaj	231	57	288	231/288	80.21%
		Povprečje					80.00%
Skupna vrednost vseh kombinacij mutacij brez ponavljanja			**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
		Skupaj	597	105	702	597/702	85.04%
		Povprečje					85.04%

Priloga 3 Heronova formula

I. Program

```
/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class HeronsFormula {

    public double heronsFormula(double a, double b, double c) {
        // If any side equals zero it is not a triangle thus the area is zero
        if (a == 0.0 || b == 0.0 || c == 0.0)
            return 0.0;

        // s = semiperimeter of the triangle
        double s = (a + b + c)/2;

        // A = area of the triangle
        double A = Math.sqrt(s*(s - a)*(s - b)*(s - c));

        return A;
    }
}
```

II. Test

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class HeronsFormulaTest {

    private HeronsFormula hf;
    private static final double DELTA = 1e-15;

    public HeronsFormulaTest() {
    }

    @Before
    public void setUp() {
        hf = new HeronsFormula();
    }

    @After
    public void tearDown() {
        hf = null;
    }

    @Test
    public void testLineLineIntersectionSideIsZero() {
        double result;
        result = hf.heronsFormula(0.0, 13.0, 15.0);
        assertEquals(0.0, result, DELTA);
        result = hf.heronsFormula(4.0, 0.0, 15.0);
        assertEquals(0.0, result, DELTA);
        result = hf.heronsFormula(4.0, 13.0, 0.0);
        assertEquals(0.0, result, DELTA);
    }

    @Test
    public void testHeronsFormula () {
        double result = hf.heronsFormula(2.0, 14.0, 14.0);
        assertEquals(13.96424004376894, result, DELTA);
    }
}
```

III. Rezultati testiranja mutantov prvega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

1. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	7	0	7	7/7	100.0%
	-	6	0	6	6/6	100.0%
	*	6	0	6	6/6	100.0%
	/	6	2	8	6/8	75.0%
	%	7	2	9	7/9	77.7%
	Skupaj	32	4	36	32/36	88.89%
	Povprečje					90.56%

2. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ -	13	0	13	13/13	100.0%
	+ *	13	0	13	13/13	100.0%
	+ /	13	2	15	13/15	86.6%
	+ %	14	2	16	14/16	87.5%
	- *	12	0	12	12/12	100.0%
	- /	12	2	14	12/14	85.7%
	- %	13	2	15	13/15	86.6%
	* /	12	2	14	12/14	85.7%
	* %	13	2	15	13/15	86.6%
	/ %	13	4	17	13/17	76.5%
	Skupaj	128	16	144	128/144	88.89%
	Povprečje					89.54%

3. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - *	19	0	19	19/19	100.0%
	+ - /	19	2	21	19/21	90.5%
	+ - %	20	2	22	20/22	90.9%
	+ * /	19	2	21	19/21	90.5%
	+ * %	20	2	22	20/22	90.9%
	+ / %	20	4	24	20/24	83.3%
	- * /	18	2	20	18/20	90.0%
	- * %	19	2	21	19/21	90.5%
	- / %	19	4	23	19/23	82.6%
	* / %	19	4	23	19/23	82.6%
	Skupaj	192	24	216	192/216	88.89%
	Povprečje					89.18%

4. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * /	25	2	27	25/27	92.6%
	+ - * %	26	2	28	26/28	92.9%
	+ - / %	26	4	30	26/30	86.7%
	+ * / %	26	4	30	26/30	86.7%
	- * / %	25	4	29	25/29	86.2%
	Skupaj	128	16	144	128/144	88.89%
	Povprečje					89.00%

5. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * / %	32	4	36	32/36	88.89%
	Povprečje					88.89%

IV. Rezultati testiranja mutantov drugega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

Vse kombinacije mutacij drugega reda	Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	+	42	0	42	42/42	100.0%
		-	32	6	38	32/38	84.2%
		*	38	0	38	38/38	100.0%
		/	50	0	50	50/50	100.0%
		%	53	3	56	53/56	94.6%
		Skupaj	215	9	224	215/224	95.98%
		Povprečje					95.77%
	-	+	32	6	38	32/38	84.2%
		-	30	0	30	30/30	100.0%
		*	33	0	33	33/33	100.0%
		/	43	0	43	43/43	100.0%
		%	47	1	48	47/48	97.9%
		Skupaj	185	7	192	185/192	96.35%
		Povprečje					96.43%
	*	+	38	0	38	38/38	100.0%
		-	33	0	33	33/33	100.0%
		*	30	0	30	30/30	100.0%
		/	43	0	43	43/43	100.0%
		%	48	0	48	48/48	100.0%
		Skupaj	192	0	192	192/192	100.00%
		Povprečje					100.00%

	/	+	50	0	50	50/50	100.0%
		-	43	0	43	43/43	100.0%
		*	43	0	43	43/43	100.0%
		/	54	2	56	54/56	96.4%
		%	60	4	64	60/64	93.8%
		Skupaj	250	6	256	250/256	97.66%
		Povprečje					98.04%
	%	+	53	3	56	53/56	94.6%
		-	47	1	48	47/48	97.9%
		*	48	0	48	48/48	100.0%
		/	60	4	64	60/64	93.8%
		%	70	2	72	70/72	97.2%
		Skupaj	278	10	288	278/288	96.53%
		Povprečje					96.71%
Skupna vrednost vseh kombinacij mutacij brez ponavljanja			**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
		Skupaj	673	18	691	673/691	97.40%
		Povprečje					97.40%

Priloga 4 Presečišče premic

I. Program

```
/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class LineLineIntersection {

    public double[] lineLineIntersection(double x1, double y1, double x2, double
                                y2, double x3, double y3, double x4, double y4) {
        double z = (x1 - x2)*(y3 - y4) - (y1 - y2)*(x3 - x4);

        if (z == 0)
            return new double[]{};

        double Px = ((x1*y2 - y1*x2)*(x3 - x4) - (x1 - x2)*(x3*y4 - y3*x4))/z;
        double Py = ((x1*y2 - y1*x2)*(y3 - y4) - (y1 - y2)*(x3*y4 - y3*x4))/z;

        return new double[]{Px, Py};
    }
}
```

II. Test

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class LineLineIntersectionTest {

    private LineLineIntersection lli;
    private static final double DELTA = 1e-15;

    public LineLineIntersectionTest() {
    }

    @Before
    public void setUp() {
        lli = new LineLineIntersection();
    }

    @After
    public void tearDown() {
        lli = null;
    }

    @Test
    public void testLineLineIntersection() {
        double[] result = lli.lineLineIntersection(-1.0, 1.0, 1.0, -1.0, -1.0, -
                                                   1.0, 1.0, 1.0);
        assertEquals(2, result.length);
        assertEquals(0.0, result[0], DELTA);
        assertEquals(0.0, result[1], DELTA);
    }

    @Test
    public void testLineLineIntersectionParallel() {
        double[] result = lli.lineLineIntersection(1.0, 2.0, 5.0, 2.0, -3.0, 7.0,
                                                   2.0, 7.0);
        assertEquals(0, result.length);
    }
}
```

III. Rezultati testiranja mutantov prvega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

1. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	21	10	31	21/31	67.7%
	-	14	2	16	14/16	87.5%
	*	6	11	17	6/17	35.3%
	/	12	17	29	12/29	41.4%
	%	14	17	31	14/31	45.2%
	Skupaj	67	57	124	67/124	54.03%
	Povprečje					55.42%

2. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ -	35	12	47	35/47	74.5%
	+ *	27	21	48	27/48	56.3%
	+ /	33	27	60	33/60	55.0%
	+ %	35	27	62	35/62	56.5%
	- *	20	13	33	20/33	60.6%
	- /	26	19	45	26/45	57.7%
	- %	28	19	47	28/47	59.6%
	* /	18	28	46	18/46	39.1%
	* %	20	28	48	20/48	41.6%
	/ %	26	34	60	26/60	43.3%
	Skupaj	268	228	496	268/496	54.03%
	Povprečje					54.43%

3. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - *	41	23	64	41/64	64.1%
	+ - /	47	29	76	47/76	61.8%
	+ - %	49	29	78	49/78	62.8%
	+ * /	39	38	77	39/77	50.6%
	+ * %	41	38	79	41/79	51.9%
	+ / %	47	44	91	47/91	51.6%
	- * /	32	30	62	32/62	51.6%
	- * %	34	30	64	34/64	53.1%
	- / %	40	36	76	40/76	52.6%
	* / %	32	45	77	32/77	41.6%
	Skupaj	402	342	744	402/744	54.03%
	Povprečje					54.18%

4. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * /	53	40	93	53/93	57.0%
	+ - * %	55	40	95	55/95	57.9%
	+ - / %	61	46	107	61/107	57.0%
	+ * / %	53	55	108	53/108	49.1%
	- * / %	46	47	93	46/93	49.5%
	Skupaj	268	228	496	268/496	54.03%
	Povprečje					54.09%

5. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * / %	67	57	124	67/124	54.03%
	Povprečje					54.03%

IV. Rezultati testiranja mutantov drugega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

Vse kombinacije mutacij drugega reda	Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	+	804	126	930	804/930	86.5%
		-	440	40	480	440/480	91.7%
		*	387	123	510	387/510	75.9%
		/	690	180	870	690/870	79.3%
		%	717	213	930	717/930	77.1%
		Skupaj	3038	682	3720	3038/3720	81.67%
		Povprečje					82.08%
	-	+	440	40	480	440/480	91.7%
		-	226	14	240	226/240	94.2%
		*	231	39	270	231/270	85.6%
		/	403	47	450	403/450	89.6%
		%	410	70	480	410/480	85.4%
		Skupaj	1710	210	1920	1710/1920	89.06%
		Povprečje					89.27%
	*	+	387	123	510	387/510	75.9%
		-	231	39	270	231/270	85.6%
		*	148	124	272	148/272	54.4%
		/	286	192	478	286/478	59.8%
		%	308	202	510	308/510	60.4%
		Skupaj	1360	680	2040	1360/2040	66.67%
		Povprečje					67.21%

	/	+	690	180	870	690/870	79.3%
		-	403	47	450	403/450	89.6%
		*	286	192	478	286/478	59.8%
		/	528	284	812	528/812	65.0%
		%	569	301	870	569/870	65.4%
		Skupaj	2476	1004	3480	2476/3480	71.15%
		Povprečje					71.83%
	%	+	717	213	930	717/930	77.1%
		-	410	70	480	410/480	85.4%
		*	308	202	510	308/510	60.4%
		/	569	301	870	569/870	65.4%
		%	620	310	930	620/930	66.7%
		Skupaj	2624	1096	3720	2624/3720	70.54%
		Povprečje					70.99%
Skupna vrednost vseh kombinacij mutacij brez ponavljanja			**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
		Skupaj	6767	2265	9032	6767/9032	74.92%
		Povprečje					74.92%

Priloga 5 Množenje matrik

I. Program

```
/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class MultiplyMatrix {

    public int[][] multiplyMatrix(int[][] mtrx1, int[][] mtrx2) {
        // Matrix is empty
        if (mtrx1 == null || mtrx2 == null || mtrx1.length == 0 || mtrx2.length ==
             0)
            return new int[][]{};

        int m = mtrx1.length;
        int n = mtrx1[0].length;
        int p = mtrx2.length;
        int q = mtrx2[0].length;

        // Matrix forms don't match
        if (n != p)
            return new int[][]{};

        int[][] product = new int[m][q];
        int sum = 0;

        for (int i = 0; i < m; i++){
            for (int j = 0; j < q; j++) {
               for (int k = 0; k < p; k++) {
                  sum = sum + mtrx1[i][k] * mtrx2[k][j];
               }
               product[i][j] = sum;
               sum = 0;
            }
         }
        return product;
    }
}
```

II. Test

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class MultiplyMatrixTest {

    private MultiplyMatrix mm;

    public MultiplyMatrixTest() {
    }

    @Before
    public void setUp() {
        mm = new MultiplyMatrix();
    }

    @After
    public void tearDown() {
        mm = null;
    }

    // Invalid input
    //-------------------------------------------------------

    @Test
    public void testMultiplyMatrixNull1() {
        int[][] mtrx1 = null;
        int[][] mtrx2 = {};
        assertEquals(0, mm.multiplyMatrix(mtrx1, mtrx2).length);
    }

    @Test
    public void testMultiplyMatrixNull2() {
        int[][] mtrx1 = {};
        int[][] mtrx2 = null;
        assertEquals(0, mm.multiplyMatrix(mtrx1, mtrx2).length);
    }

    @Test
    public void testMultiplyMatrixEmpty1() {
        int[][] mtrx1 = {};
        int[][] mtrx2 = { {1, 2, 3},
                          {4, 5, 6},
                          {7, 8, 9} };
        assertEquals(0, mm.multiplyMatrix(mtrx1, mtrx2).length);
    }

    @Test
    public void testMultiplyMatrixEmpty2() {
```

```
        int[][] mtrx1 = { {1, 2, 3},
                          {4, 5, 6},
                          {7, 8, 9} };
        int[][] mtrx2 = {};
        assertEquals(0, mm.multiplyMatrix(mtrx1, mtrx2).length);
    }

    @Test
    public void testMultiplyMatrixNotCompatable() {
        int[][] mtrx1 = { {1, 2, 3},
                          {4, 5, 6},
                          {7, 8, 9} };
        int[][] mtrx2 = { {1, 2},
                          {3, 4} };
        assertEquals(0, mm.multiplyMatrix(mtrx1, mtrx2).length);
    }

    // Multiplication
    //-------------------------------------------------------

    @Test
    public void testMultiplyMatrix() {
        int[][] mtrx1 = { {1, 2, 3},
                          {4, 5, 6} };
        int[][] mtrx2 = { {1, 1},
                          {1, 1},
                          {1, 1} };
        int[][] expect = {{6, 6},
                          {15, 15}};
        assertEquals(expect, mm.multiplyMatrix(mtrx1, mtrx2));
    }
}
```

III. Rezultati testiranja mutantov prvega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

1. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	1	0	1	1/1	100.0%
	-	2	0	2	2/2	100.0%
	*	1	0	1	1/1	100.0%
	/	1	1	2	1/2	50.0%
	%	2	0	2	2/2	100.0%
	Skupaj	7	1	8	7/8	87.50%
	Povprečje					90.00%

2. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ -	3	0	3	3/3	100.0%
	+ *	2	0	2	2/2	100.0%
	+ /	2	1	3	2/3	66.6%
	+ %	3	0	3	3/3	100.0%
	- *	3	0	3	3/3	100.0%
	- /	3	1	4	3/4	75.0%
	- %	4	0	4	4/4	100.0%
	* /	2	1	3	2/3	66.6%
	* %	3	0	3	3/3	100.0%
	/ %	3	1	4	3/4	75.0%
	Skupaj	28	4	32	28/32	87.50%
	Povprečje					88.33%

3. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - *	4	0	4	4/4	100.0%
	+ - /	4	1	5	4/5	80.0%
	+ - %	5	0	5	5/5	100.0%
	+ * /	3	1	4	3/4	75.0%
	+ * %	4	0	4	4/4	100.0%
	+ / %	4	1	5	4/5	80.0%
	- * /	4	1	5	4/5	80.0%
	- * %	5	0	5	5/5	100.0%
	- / %	5	1	6	5/6	83.3%
	* / %	4	1	5	4/5	80.0%
	Skupaj	42	6	48	42/48	87.50%
	Povprečje					87.83%

4. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * /	5	1	6	5/6	83.3%
	+ - * %	6	0	6	6/6	100.0%
	+ - / %	6	1	7	6/7	85.7%
	+ * / %	5	1	6	5/6	83.3%
	- * / %	6	1	7	6/7	85.7%
	Skupaj	28	4	32	28/32	87.50%
	Povprečje					87.62%

5. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * / %	7	1	8	7/8	87.50%
	Povprečje					87.50%

IV. Rezultati testiranja mutantov drugega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

Vse kombinacije mutacij drugega reda	Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	+	0	0	0	0/0	0.0%
		-	1	0	1	1/1	100.0%
		*	1	0	1	1/1	100.0%
		/	1	0	1	1/1	100.0%
		%	1	0	1	1/1	100.0%
		Skupaj	4	0	4	4/4	100.00%
		Povprečje					100.00%
	-	+	1	0	1	1/1	100.0%
		-	2	0	2	2/2	100.0%
		*	1	0	1	1/1	100.0%
		/	2	0	2	2/2	100.0%
		%	2	0	2	2/2	100.0%
		Skupaj	8	0	8	8/8	100.00%
		Povprečje					100.00%
	*	+	1	0	1	1/1	100.0%
		-	1	0	1	1/1	100.0%
		*	0	0	0	0/0	0.0%
		/	1	0	1	1/1	100.0%
		%	1	0	1	1/1	100.0%
		Skupaj	4	0	4	4/4	100.00%
		Povprečje					100.00%

	/	+	1	0	1	1/1	100.0%
		-	2	0	2	2/2	100.0%
		*	1	0	1	1/1	100.0%
		/	2	0	2	2/2	100.0%
		%	2	0	2	2/2	100.0%
		Skupaj	8	0	8	8/8	100.00%
		Povprečje					100.00%
	%	+	1	0	1	1/1	100.0%
		-	2	0	2	2/2	100.0%
		*	1	0	1	1/1	100.0%
		/	2	0	2	2/2	100.0%
		%	2	0	2	2/2	100.0%
		Skupaj	8	0	8	8/8	100.00%
		Povprečje					100.00%
Skupna vrednost vseh kombinacij mutacij brez ponavljanja			**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
		Skupaj	19	0	19	19/19	100.00%
		Povprečje					100.00%

Priloga 6 Razdalja točke od premice

I. Program

```
/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class PointToLineDistance {

    public double pointToLineDistance(double a, double b, double c, double x,
                                      double y) {
        // Invalid line
        if (a == 0 && b == 0)
            throw new IllegalArgumentException();

        double d = Math.abs(a*x + b*y + c)/Math.sqrt(Math.pow(a, 2) + Math.pow(b,
                   2));

        return d;
    }
}
```

II. Test

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class PointToLineDistanceTest {

    private PointToLineDistance ptld;
    private static final double DELTA = 1e-15;

    public PointToLineDistanceTest() {
    }

    @Before
    public void setUp() {
        ptld = new PointToLineDistance();
    }

    @After
    public void tearDown() {
        ptld = null;
    }

    @Test(expected=IllegalArgumentException.class)
    public void testPointToLineDistanceInvalidLine() {
        ptld.pointToLineDistance(0.0, 0.0, 5.0, 3.0, -10.0);
    }

    @Test
    public void testPointToLineDistance () {
        double result = ptld.pointToLineDistance(1.0, 1.0, 4.0, 1.0, 1.0);
        assertEquals(6.0/Math.sqrt(2), result, DELTA);
    }
}
```

III. Rezultati testiranja mutantov prvega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

1. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	3	0	3	3/3	100.0%
	-	6	0	6	6/6	100.0%
	*	4	0	4	4/4	100.0%
	/	3	2	5	3/5	60.0%
	%	6	0	6	6/6	100.0%
	Skupaj	22	2	24	22/24	91.67%
	Povprečje					92.00%

2. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ -	9	0	9	9/9	100.0%
	+ *	7	0	7	7/7	100.0%
	+ /	6	2	8	6/8	75.0%
	+ %	9	0	9	9/9	100.0%
	- *	10	0	10	10/10	100.0%
	- /	9	2	11	9/11	81.8%
	- %	12	0	12	12/12	100.0%
	* /	7	2	9	7/9	77.8%
	* %	10	0	10	10/10	100.0%
	/ %	9	2	11	9/11	81.8%
	Skupaj	88	8	96	88/96	91.67%
	Povprečje					91.64%

3. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - *	13	0	13	13/13	100.0%
	+ - /	12	2	14	12/14	85.7%
	+ - %	15	0	15	15/15	100.0%
	+ * /	10	2	12	10/12	83.3%
	+ * %	13	0	13	13/13	100.0%
	+ / %	12	2	14	12/14	85.7%
	- * /	13	2	15	13/15	86.7%
	- * %	16	0	16	16/16	100.0%
	- / %	15	2	17	15/17	88.2%
	* / %	13	2	15	13/15	86.7%
	Skupaj	132	12	144	132/144	91.67%
	Povprečje					91.63%

4. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * /	16	2	18	16/18	88.9%
	+ - * %	19	0	19	19/19	100.0%
	+ - / %	18	2	20	18/20	90.0%
	+ * / %	16	2	18	16/18	88.9%
	- * / %	19	2	21	19/21	90.5%
	Skupaj	88	8	96	88/96	91.67%
	Povprečje					91.65%

5. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * / %	22	2	24	22/24	91.67%
	Povprečje					91.67%

IV. Rezultati testiranja mutantov drugega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

Vse kombinacije mutacij drugega reda	Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	+	6	0	6	6/6	100.0%
		-	13	2	15	13/15	86.7%
		*	9	2	11	9/11	81.8%
		/	11	2	13	11/13	84.6%
		%	13	2	15	13/15	86.7%
		Skupaj	52	8	60	52/60	86.67%
		Povprečje					87.95%
	-	+	13	2	15	13/15	86.7%
		-	30	0	30	30/30	100.0%
		*	20	0	20	20/20	100.0%
		/	25	0	25	25/25	100.0%
		%	30	0	30	30/30	100.0%
		Skupaj	118	2	120	118/120	98.33%
		Povprečje					97.33%
	*	+	9	2	11	9/11	81.8%
		-	20	0	20	20/20	100.0%
		*	12	0	12	12/12	100.0%
		/	17	0	17	17/17	100.0%
		%	20	0	20	20/20	100.0%
		Skupaj	78	2	80	78/80	97.50%
		Povprečje					96.36%

	/	+	11	2	13	11/13	84.6%
		-	25	0	25	25/25	100.0%
		*	17	0	17	17/17	100.0%
		/	18	2	20	18/20	90.0%
		%	25	0	25	25/25	100.0%
		Skupaj	96	4	100	96/100	96.00%
		Povprečje					94.92%
	%	+	13	2	15	13/15	86.7%
		-	30	0	30	30/30	100.0%
		*	20	0	20	20/20	100.0%
		/	25	0	25	25/25	100.0%
		%	30	0	30	30/30	100.0%
		Skupaj	118	2	120	118/120	98.33%
		Povprečje					97.33%
Skupna vrednost vseh kombinacij mutacij brez ponavljanja			**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
		Skupaj	279	10	289	279/289	96.54%
		Povprečje					96.54%

Priloga 7 Ničle kvadratne funkcije

I. Program

```
/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class QuadFuncRoots {

    public double[] quadFuncRoots(double a, double b, double c) {
        double discriminant = Math.pow(b, 2) - 4*a*c;

        // If the discriminant is positive, the function has two (real) roots -
             the graph intersects the x axis in two points
        // + 0.0 gets rid of the negative zero (-0.0)
        if (discriminant > 0) {
            double x1 = (-b + Math.sqrt(discriminant))/(2*a);
            double x2 = (-b - Math.sqrt(discriminant))/(2*a);
            return new double[]{x1, x2};

        }
        // If the discriminant is zero, the function has one (real) root - the
             graph touches the x axis in a single point
        else if (discriminant == 0) {
            double x = (-b)/(2*a);
            return new double[]{x};
        }
        // If the discriminant is negative, the function doesn't have any (real)
             roots - the graph doesn't intersect the x axis in any point
        else {
            return new double[]{};
        }
    }
}
```

II. Test

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class QuadFuncRootsTest {

    private QuadFuncRoots qfr;
    private static final double DELTA = 1e-15;

    public QuadFuncRootsTest() {
    }

    @Before
    public void setUp() {
        qfr = new QuadFuncRoots();
    }

    @After
    public void tearDown() {
        qfr = null;
    }

    // No roots
    // ------------------------------------------------------
    @Test
    public void testQuadFuncRootsZeroNeg() {
      double[] result = qfr.quadFuncRoots(1.0, 4.0, 5.0);
      assertEquals(0, result.length);
    }

    // One root
    // ------------------------------------------------------

    @Test
    public void testQuadFuncRootsSingle() {
      double[] result = qfr.quadFuncRoots(1.0, 0.0, 0.0);
      assertEquals(1, result.length);
      assertEquals(0.0, result[0], DELTA);
    }

    @Test
    public void testQuadFuncRootsLengthSingle() {
      double[] result = qfr.quadFuncRoots(1.0, 0.0, 0.0);
      assertEquals(1, result.length);
    }
```

```
    // Two roots
    // ------------------------------------------------------

    @Test
    public void testQuadFuncRootsDouble() {
       double[] result = qfr.quadFuncRoots(1.0, -2.0, -3.0);
       assertEquals(2, result.length);
       assertEquals(3.0, result[0], DELTA);
       assertEquals(-1.0, result[1], DELTA);
    }

    @Test
    public void testQuadFuncRootsLengthDouble() {
       double[] result = qfr.quadFuncRoots(1.0, -2.0, -3.0);
       assertEquals(2, result.length);
    }
}
```

III. Rezultati testiranja mutantov prvega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

1. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	9	1	10	9/10	90.0%
	-	8	1	9	8/9	88.9%
	*	5	1	6	5/6	83.3%
	/	4	4	8	4/8	50.0%
	%	10	1	11	10/11	90.9%
	Skupaj	36	8	44	36/44	81.82%
	Povprečje					80.63%

2. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ -	17	2	19	17/19	89.5%
	+ *	14	2	16	14/16	87.5%
	+ /	13	5	18	13/18	72.2%
	+ %	19	2	21	19/21	90.5%
	- *	13	2	15	13/15	86.7%
	- /	12	5	17	12/17	70.6%
	- %	18	2	20	18/20	90.0%
	* /	9	5	14	9/14	64.3%
	* %	15	2	17	15/17	88.2%
	/ %	14	5	19	14/19	73.7%
	Skupaj	144	32	176	144/176	81.82%
	Povprečje					81.31%

3. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - *	22	3	25	22/25	88.0%
	+ - /	21	6	27	21/27	77.8%
	+ - %	27	3	30	27/30	90.0%
	+ * /	18	6	24	18/24	75.0%
	+ * %	24	3	27	24/27	88.9%
	+ / %	23	6	29	23/29	79.3%
	- * /	17	6	23	17/23	73.9%
	- * %	23	3	26	23/26	88.5%
	- / %	22	6	28	22/28	78.6%
	* / %	19	6	25	19/25	76.0%
	Skupaj	216	48	264	216/264	81.82%
	Povprečje					81.59%

4. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * /	26	7	33	26/33	78.8%
	+ - * %	32	4	36	32/36	88.9%
	+ - / %	31	7	38	31/38	81.6%
	+ * / %	28	7	35	28/35	80.0%
	- * / %	27	7	34	27/34	79.4%
	Skupaj	144	32	176	144/176	81.82%
	Povprečje					81.73%

5. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * / %	36	8	44	36/44	81.82%
	Povprečje					81.82%

IV. Rezultati testiranja mutantov drugega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

Vse kombinacije mutacij drugega reda	Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	+	90	0	90	90/90	100.0%
		-	80	2	82	80/82	97.6%
		*	54	1	55	54/55	98.2%
		/	70	3	73	70/73	95.9%
		%	98	2	100	98/100	98.0%
		Skupaj	392	8	400	392/400	98.00%
		Povprečje					97.93%
	-	+	80	2	82	80/82	97.6%
		-	71	1	72	71/72	98.6%
		*	49	1	50	49/50	98.0%
		/	63	3	66	63/66	95.5%
		%	88	2	90	88/90	97.8%
		Skupaj	351	9	360	351/360	97.50%
		Povprečje					97.48%
	*	+	54	1	55	54/55	98.2%
		-	49	1	50	49/50	98.0%
		*	30	0	30	30/30	100.0%
		/	41	4	45	41/45	91.1%
		%	59	1	60	59/60	98.3%
		Skupaj	233	7	240	233/240	97.08%
		Povprečje					97.13%

	/	+	70	3	73	70/73	95.9%
		-	63	3	66	63/66	95.5%
		*	41	4	45	41/45	91.1%
		/	44	12	56	44/56	78.6%
		%	76	4	80	76/80	95.0%
		Skupaj	294	26	320	294/320	91.88%
		Povprečje					91.21%
	%	+	98	2	100	98/100	98.0%
		-	88	2	90	88/90	97.8%
		*	59	1	60	59/60	98.3%
		/	76	4	80	76/80	95.0%
		%	110	0	110	110/110	100.0%
		Skupaj	431	9	440	431/440	97.95%
		Povprečje					97.82%
Skupna vrednost vseh kombinacij mutacij brez ponavljanja			**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
		Skupaj	1023	36	1059	1023/1059	96.60%
		Povprečje					96.60%

Priloga 8 Strassenovo množenje matrik

I. Program

```
import java.util.ArrayList;

/**
 * http://martin-thoma.com/strassen-algorithm-in-python-java-cpp/#tocAnchor-1-4
 * https://github.com/MartinThoma/matrix-multiplication/tree/master/Java
 * @author Martin Thoma, January 23rd, 2013
 * @title The Strassen algorithm in Python, Java and C++
 */
public class StrassenAlgorithm {

    static int LEAF_SIZE = 1;

    public static int[][] ikjAlgorithm(int[][] A, int[][] B) {
        int n = A.length;

        // initialise C
        int[][] C = new int[n][n];

        for (int i = 0; i < n; i++) {
            for (int k = 0; k < n; k++) {
                for (int j = 0; j < n; j++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        return C;
    }

    private static int[][] add(int[][] A, int[][] B) {
        int n = A.length;
        int[][] C = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                C[i][j] = A[i][j] + B[i][j];
            }
        }
        return C;
    }

    private static int[][] subtract(int[][] A, int[][] B) {
        int n = A.length;
        int[][] C = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                C[i][j] = A[i][j] - B[i][j];
            }
        }
        return C;
    }
```

```
    private static int nextPowerOfTwo(int n) {
        int log2 = (int) Math.ceil(Math.log(n) / Math.log(2));
        return (int) Math.pow(2, log2);
    }

    public static int[][] strassen(ArrayList<ArrayList<Integer>> A,
ArrayList<ArrayList<Integer>> B) {
        // Make the matrices bigger so that you can apply the strassen
        // algorithm recursively without having to deal with odd
        // matrix sizes
        int n = A.size();
        int m = nextPowerOfTwo(n);
        int[][] APrep = new int[m][m];
        int[][] BPrep = new int[m][m];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                APrep[i][j] = A.get(i).get(j);
                BPrep[i][j] = B.get(i).get(j);
            }
        }

        int[][] CPrep = strassenR(APrep, BPrep);
        int[][] C = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                C[i][j] = CPrep[i][j];
            }
        }
        return C;
    }

    private static int[][] strassenR(int[][] A, int[][] B) {
        int n = A.length;

        if (n <= LEAF_SIZE) {
            return ikjAlgorithm(A, B);
        } else {
            // initializing the new sub-matrices
            int newSize = n / 2;
            int[][] a11 = new int[newSize][newSize];
            int[][] a12 = new int[newSize][newSize];
            int[][] a21 = new int[newSize][newSize];
            int[][] a22 = new int[newSize][newSize];

            int[][] b11 = new int[newSize][newSize];
            int[][] b12 = new int[newSize][newSize];
            int[][] b21 = new int[newSize][newSize];
            int[][] b22 = new int[newSize][newSize];

            int[][] aResult = new int[newSize][newSize];
            int[][] bResult = new int[newSize][newSize];

            // dividing the matrices in 4 sub-matrices:
            for (int i = 0; i < newSize; i++) {
                for (int j = 0; j < newSize; j++) {
                    a11[i][j] = A[i][j]; // top left
                    a12[i][j] = A[i][j + newSize]; // top right
```

```
            a21[i][j] = A[i + newSize][j]; // bottom left
            a22[i][j] = A[i + newSize][j + newSize]; // bottom right

            b11[i][j] = B[i][j]; // top left
            b12[i][j] = B[i][j + newSize]; // top right
            b21[i][j] = B[i + newSize][j]; // bottom left
            b22[i][j] = B[i + newSize][j + newSize]; // bottom right
        }
    }

    // Calculating p1 to p7:
    aResult = add(a11, a22);
    bResult = add(b11, b22);
    int[][] p1 = strassenR(aResult, bResult);
    // p1 = (a11+a22) * (b11+b22)

    aResult = add(a21, a22); // a21 + a22
    int[][] p2 = strassenR(aResult, b11); // p2 = (a21+a22) * (b11)

    bResult = subtract(b12, b22); // b12 - b22
    int[][] p3 = strassenR(a11, bResult);
    // p3 = (a11) * (b12 - b22)

    bResult = subtract(b21, b11); // b21 - b11
    int[][] p4 = strassenR(a22, bResult);
    // p4 = (a22) * (b21 - b11)

    aResult = add(a11, a12); // a11 + a12
    int[][] p5 = strassenR(aResult, b22);
    // p5 = (a11+a12) * (b22)

    aResult = subtract(a21, a11); // a21 - a11
    bResult = add(b11, b12); // b11 + b12
    int[][] p6 = strassenR(aResult, bResult);
    // p6 = (a21-a11) * (b11+b12)

    aResult = subtract(a12, a22); // a12 - a22
    bResult = add(b21, b22); // b21 + b22
    int[][] p7 = strassenR(aResult, bResult);
    // p7 = (a12-a22) * (b21+b22)

    // calculating c21, c21, c11 e c22:
    int[][] c12 = add(p3, p5); // c12 = p3 + p5
    int[][] c21 = add(p2, p4); // c21 = p2 + p4

    aResult = add(p1, p4); // p1 + p4
    bResult = add(aResult, p7); // p1 + p4 + p7
    int[][] c11 = subtract(bResult, p5);
    // c11 = p1 + p4 - p5 + p7

    aResult = add(p1, p3); // p1 + p3
    bResult = add(aResult, p6); // p1 + p3 + p6
    int[][] c22 = subtract(bResult, p2);
    // c22 = p1 + p3 - p2 + p6

    // Grouping the results obtained in a single matrix:
    int[][] C = new int[n][n];
```

```
            for (int i = 0; i < newSize; i++) {
                for (int j = 0; j < newSize; j++) {
                    C[i][j] = c11[i][j];
                    C[i][j + newSize] = c12[i][j];
                    C[i + newSize][j] = c21[i][j];
                    C[i + newSize][j + newSize] = c22[i][j];
                }
            }
            return C;
        }
    }
}
```

II. Test

```
import java.util.ArrayList;
import java.util.Arrays;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Jure Cetina, oktober 2014
 */
public class StrassenAlgorithmTest {

    private StrassenAlgorithm sa;

    public StrassenAlgorithmTest() {
    }

    @Before
    public void setUp() {
        sa = new StrassenAlgorithm();
    }

    @After
    public void tearDown() {
        sa = null;
    }

    @Test
    public void testStrassen() {
        Integer[] mtrxrow = {1, 1, 1, 1};
        ArrayList<Integer> row = new ArrayList<>();
        row.addAll(Arrays.asList(mtrxrow));

        // Matrix 1
        ArrayList<ArrayList<Integer>> mtrx1 = new ArrayList<>();
        mtrx1.add(row); mtrx1.add(row); mtrx1.add(row); mtrx1.add(row);

        // Matrix 2
        ArrayList<ArrayList<Integer>> mtrx2 = new ArrayList<>();
        mtrx2.add(row); mtrx2.add(row); mtrx2.add(row); mtrx2.add(row);

        // Expected
        int[][] expect = {{4, 4, 4, 4},
                          {4, 4, 4, 4},
                          {4, 4, 4, 4},
                          {4, 4, 4, 4}};

        assertEquals(expect, sa.strassen(mtrx1, mtrx2));
    }
}
```

III. Rezultati testiranja mutantov prvega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

1. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	3	1	4	3/4	75.0%
	-	16	0	16	16/16	100.0%
	*	8	8	16	8/16	50.0%
	/	7	8	15	7/15	46.7%
	%	9	8	17	9/17	52.9%
	Skupaj	43	25	68	43/68	63.24%
	Povprečje					64.92%

2. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ -	19	1	20	19/20	95.0%
	+ *	11	9	20	11/20	55.0%
	+ /	10	9	19	10/19	52.6%
	+ %	12	9	21	12/21	57.1%
	- *	24	8	32	24/32	75.0%
	- /	23	8	31	23/31	74.2%
	- %	25	8	33	25/33	75.8%
	* /	15	16	31	15/31	48.4%
	* %	17	16	33	17/33	51.5%
	/ %	16	16	32	16/32	50.0%
	Skupaj	172	100	272	172/272	63.24%
	Povprečje					63.46%

3. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - *	27	9	36	27/36	75.0%
	+ - /	26	9	35	26/35	74.3%
	+ - %	28	9	37	28/37	75.7%
	+ * /	18	17	35	18/35	51.4%
	+ * %	20	17	37	20/37	54.1%
	+ / %	19	17	36	19/36	52.8%
	- * /	31	16	47	31/47	66.0%
	- * %	33	16	49	33/49	67.3%
	- / %	32	16	48	32/48	66.7%
	* / %	24	24	48	24/48	50.0%
	Skupaj	258	150	408	258/408	63.24%
	Povprečje					63.32%

4. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * /	34	17	51	34/51	66.7%
	+ - * %	36	17	53	36/53	67.9%
	+ - / %	35	17	52	35/52	67.3%
	+ * / %	27	25	52	27/52	51.9%
	- * / %	40	24	64	40/64	62.5%
	Skupaj	172	100	272	172/272	63.24%
	Povprečje					63.26%

5. Stopnja	Mutacije	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+ - * / %	43	25	68	43/68	63.24%
	Povprečje					63.24%

IV. Rezultati testiranja mutantov drugega reda

Najmanjše število mutantov z najnižjo mutacijsko oceno

Najmanjše število mutantov z najvišjo mutacijsko oceno

Vse kombinacije mutacij drugega reda	Mutacije prvega reda	Mutacije drugega reda	Št. ubitih mutantov	Št. živih mutantov	Št. vseh mutantov	Št. ubitih mutantov/ št. vseh mutantov	Mutacijska ocena
	+	+	6	0	6	6/6	100.0%
		-	54	0	54	54/54	100.0%
		*	55	0	55	55/55	100.0%
		/	42	8	50	42/50	84.0%
		%	47	8	55	47/55	85.5%
		Skupaj	204	16	220	204/220	92.73%
		Povprečje					93.89%
	-	+	54	0	54	54/54	100.0%
		-	198	0	198	198/198	100.0%
		*	200	0	200	200/200	100.0%
		/	183	0	183	183/183	100.0%
		%	200	0	200	200/200	100.0%
		Skupaj	835	0	835	835/835	100.00%
		Povprečje					100.00%
	*	+	55	0	55	55/55	100.0%
		-	200	0	200	200/200	100.0%
		*	142	56	198	142/198	71.7%
		/	125	56	181	125/181	69.1%
		%	142	56	198	142/198	71.7%
		Skupaj	664	168	832	664/832	79.81%
		Povprečje					82.50%

	/	+	42	8	50	42/50	84.0%
		-	183	0	183	183/183	100.0%
		*	125	56	181	125/181	69.1%
		/	115	56	171	115/171	67.3%
		%	133	56	189	133/189	70.4%
		Skupaj	598	176	774	598/774	77.26%
		Povprečje					78.14%
	%	+	47	8	55	47/55	85.5%
		-	200	0	200	200/200	100.0%
		*	142	56	198	142/198	71.7%
		/	133	56	189	133/189	70.4%
		%	158	56	214	158/214	73.8%
		Skupaj	680	176	856	680/856	79.44%
		Povprečje					80.28%
Skupna vrednost vseh kombinacij mutacij brez ponavljanja			**Št. ubitih mutantov**	**Št. živih mutantov**	**Št. vseh mutantov**	**Št. ubitih mutantov/ št. vseh mutantov**	**Mutacijska ocena**
		Skupaj	1800	352	2152	1800/2152	83.64%
		Povprečje					83.64%